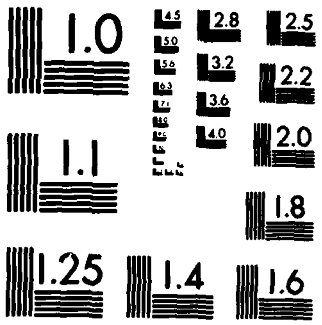
AD-A146 584 A STANDARD OPERATING SYSTEM INTERFACE FOR MICROCOMPUTER 1/2
SOFTWARE DEVELOPMENT(U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA R STEMP MAR 84

UNCLASSIFIED F/G 9/2 NL



OPY RESOLUTION TEST CHART

C

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A STANDARD OPERATING SYSTEM INTERFACE FOR MICROCOMPUTER SOFTWARE DEVELOPMENT

by

Roger Stemp

March 1984

Thesis Advisor: Daniel L. Davis

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD A146 584	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Standard Operating System Interface for Microcomputer Software Development		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis March 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Roger Stemp		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE March 1984
		13. NUMBER OF PAGES 116
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release, Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Interface, Parameter Passing, Operating System Interface, Kernel, Protocol

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The majority of discussion directed at standardizing microcomputer operating system has revolved primarily around establishment of a set of standardized primitives (a kernel) to be made available for use by programmers. To this end little progress has been made. Establishment of a universal kernel for microcomputer operating systems, or for mini or mainframes for that matter, is not only virtually impossible but also highly narrow in scope.

(continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (Continued)

This thesis presents a possible solution to standardization efforts through implementation of a 'Dynamic Kernel' achieved by the establishment of a universal protocol between application programs and microcomputer operating systems via a standard interface structure. A high level design of the necessary interface structure and recommended primitives for initial inclusion in the 'Dynamic Kernel' are presented along with brief discussions of the inherent dangers and benefits that may be encountered.

A Standard Operating System Interface for Microcomputer Software Development

by

Roger Stemp
Lieutenant, United States Navy
B.S., Auburn University, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1984

A-1

Author: [illegible]

Approved by: Daniel Davis
Thesis Advisor

[illegible]
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

The majority of discussion directed at standardizing microcomputer operating systems has revolved primarily around establishment of a set of standardized primitives (a kernel) to be made available for use by programmers. To this end little progress has been made. Establishment of a universal kernel for microcomputer operating systems, or for mini or mainframes for that matter, is not only virtually impossible but also highly narrow in scope.

This thesis presents a possible solution to standardization efforts through implementation of a 'Dynamic Kernel' achieved by the establishment of a universal protocol between application programs and microcomputer operating systems via a standard interface structure. A high level design of the necessary interface structure and recommended primitives for initial inclusion in the 'Dynamic Kernel' are presented along with brief discussions of the inherent dangers and benefits that may be encountered.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Since 1972 when Intel Corporation released the 8080 microprocessor and Motorola Corporation released the 6800 microprocessor, the microcomputer industry has experienced a rate of growth unparalleled by any other modern day industry. It is anticipated that this present growth rate will continue steadily as a greater proportion of the general population achieves computer literacy.

When microcomputers were initially introduced into the public marketplace, they were purchased primarily by computer hobbyists who possessed highly technical knowledge concerning the microprocessor's construction and operation. Today, however, microcomputers are being purchased by people with limited technical skills, a fact which is a direct result of the great influx of application software in the microcomputer marketplace. This growth in application software has made the benefits of the computer more apparent to the general public and, as a result, it has served to increase the demand for a broader spectrum of application software. Additionally, an increasing number of non-technical software designers, armed only with advanced program language skills and varying degrees of professional skills, necessitates improved man/machine interfacing.

The role of the operating system is to manage memory and other resources. Earlier operating systems for the micro-computer, hereto referred to as the personal computer, were constrained by memory limitations and were often required to fit into a two kilobyte (or less) portion of main memory. As memory constraints have diminished, operating systems for the personal computer have grown both in sophistication and size and have begun to take on close similarities to their mainframe counterparts.

The number of operating systems available for personal computers has grown substantially and it has only been recently that the subject of standardization has been addressed. This subject is one which creates a great deal of heated discussion among the numerous operating system designers as each designer has his/her own perception of what the future holds for the personal computer, as well as their future advantage in the marketplace. Standardization, many feel, can only inhibit future development. But while the debate continues, software development is impeded by the lack of direct and easy access to the operating system and machine primitives required by application software developers increasingly entering the personal computer market.

B. PURPOSE

The primary purpose of this thesis is to offer a viable solution to the current standardization question through presentation of a conceptual interface model and its associated primitives.

C. SCOPE

The scope of this thesis includes a brief survey of two existing operating systems designed specifically for the personal computer. This survey results in a collection of user-accessible primitives which contain some elements common to both as well as additional functions that have been included to aid in application software programming for the microcomputer. (See appendices.)

In an effort to enhance application program portability, a conceptual description of an operating system interface that facilitates access to this collection of primitives via a standardized protocol is presented along with its associated design considerations. A discussion of the motivation for creating a standard interface and of the inherent advantages and disadvantages of implementing such a standard is included. However, in order to constrain the scope of this thesis, many issues are not addressed and those issues which are discussed are not covered in depth.

Although actual coding of the interface is not included, an explanation of both the conceptual and possible physical characteristics is provided in sufficient depth that coding of the interface would require only moderate effort.

Finally, falling within the scope of this thesis is a discussion of recommended implementation methods, recommendations for future interface enhancements and related research.

II. INTERFACING CONSIDERATIONS

A. MOTIVATION FOR INTERFACE

Operating systems, whether designed for implementation on a microcomputer or large mainframe, perform many services other than I/O management. However, we will confine our description of the operating system to the set of functions dealing with I/O that are generally the only operating system functions that the application programmer may wish to access.

Microcomputer operating systems input/output functions, without exception, are based upon a kernel concept. This kernel in general consists of a small set of explicit hardware dependent services, commonly called Basic Input/Output Services (BIOS), in combination with a modest number of higher level services (BDOS) which are accessible to the programmer and which directly utilize the lower level BIOS functions. In most instances this select group of I/O routines is either stored in ROM, as it is in IBM's PC-DOS and Microsoft's MS-DOS, or it is included in the static portion of the operating system which remains in a fixed location in memory after system booting, as it is in Digital Research's CP/M 80. The remaining operating system services (i.e., command processor, system utilities) are either transient in memory or remain as external library (or subroutine) calls found on an external device, such as a disk drive or cache memory device.

For application programmers, these I/O primitives provide the major interface between the application program and the particular system upon which the program is being implemented. They are frequently accessed because the vast majority of programming languages do not provide I/O routines which are adequate or fast enough for real time applications. The limited number of language supplied I/O services, the inconsistent methods of invoking OS supplied primitives, and the failure of the majority of operating systems and languages to include sufficient routines for utilization of diversified output devices (e.g., plasma displays, bit mapped graphic displays, etc.) force application programmers either to design slow and inefficient programs, which are limited in their ability to display and interact with the user, or to access the necessary functions through hardware-dependent calls (such as 'poking' values in specific memory locations).

The major philosophy behind the limited number of available I/O primitives is based upon the necessity of keeping the resident portion of microcomputer operating systems as small as possible. This requirement comes from constraints that were imposed upon designers when there was a limited amount of system memory available for use by both the operating system and the application programs. However, today these constraints are less significant due to the increased memory found in today's microcomputer systems. Yet, many designers of microcomputer operating systems still have not broken away from this early philosophy, which indirectly encourages unneccesary violations of system independent software design by application programmers.

Creation of a comprehensive set of I/O primitives would reduce the need for system dependent hardware calls and it would also improve programmer productivity. The latter results from the fact that, today, most software is intensely display-oriented and highly user-interactive. The lack of primitives available to accommodate these features results in a substantial increase in program code. System dependent calls require sophisticated technical knowledge about the hardware on the part of application programmer and frequently involve complex coding to accomplish the desired result. By eliminating the necessity of such calls and by providing adequate error checking features, it is not difficult to see that programmer productivity would be significantly increased.

In all fairness to microcomputer operating system designers, it should be mentioned that some of the more recent microcomputer operating systems have tried to meet the demands of application programmers. Several of the more advanced operating systems, such as Concurrent CP/M and MS-Dos Version 2.0, do indeed provide access to a greater variety of display oriented functions; however, invocation of these primitives is generally accomplished at the assembly language level and, therefore, require additional skills of programmers. The problem, then, appears to be not only a lack of available primitives but also that access to these primitives is not easily provided in high level languages.

B. IMPACT ON LANGUAGE AND O/S DESIGN

High level languages, with few exceptions, are far from standardized. This is due, in part, to the inherent inadequacies present in many languages for providing sufficient I/O services and in part to the diversity of the methods used to invoke external function calls from within a given language. Invoking external code from within a high level language itself is a highly arbitrary and unsettled matter. The net result is language modification by language implementors attempting to compensate for these weaknesses thereby ultimately destroying source code portability. The solution to these problems appears to be establishment of a universal protocol for accessing external primitives and acceptance by operating system designers that the host system should assume complete responsibility for providing a comprehensive set of I/O functions.

This last issue may impact on current operating system design philosophies and possibly future language development as it would require shifting a large portion of the responsibility for providing adequate I/O processes from the language domain to that of the operating systems. This does not mean, however, that present langauge I/O functions should be abandoned, but rather that an alternate method be established for providing these services.

An undesirable, but inevitable, side effect in this shift would be an increased burden upon the application programmer since more stringent programming practices would be required in order to avoid potential pitfalls. However, the advantages gained in terms of interactive display flexibility and increased programmer productivity may very well outweigh the possible disadvantages.

Permitting extensive use of I/O processing which is external to the application language generates several advantages and disadvantages, not just from the viewpoint of application programmers but also from the viewpoint of accepted language design principles. A brief summary of some of the more obvious issues is listed below.

1. <u>Disadvantages</u>

a. Degradation of Typing Control Mechanisms

Parameter passing and data exchange may lead to either intentional or unintentional circumvention of data typing control mechanisms. The burden for ensuring that this does not occur will be placed upon the application programmer.

b. Possible Loss of Data Integrity

Several of the primitives which will be recommended for inclusion in a prototype interface permit massive block movement of data; the possible result is that some areas containing critical data could be overwritten. Once again, the responsibility for ensuring that this does not occur will be placed in the hands of the programmer.

c. Degradation of Code Readability

Excessive invocation of external I/O requests may result in a breakdown in the readability of source code; however, through adequate documentation within the source code this may not be a significant problem. In fact, it may be a blessing in disguise since a large portion of the program code, which was originally dedicated to complex I/O processing, may be eliminated thereby improving understandability of the overall program logic.

d. Loss of Debugging Capability

Since many I/O requests may no longer be within control of the language itself, compile time, syntax errors and run-time boundary value errors will not be readily identifible. These negative aspects can be partially eliminated through the use of a precompiler furnished as a system utility and through thoughtful error handling analysis by OS designers.

2. <u>Advantages</u>

a. Increase in Language Portability

Reducing the temptation of language implementors to add unnecessary frills designed to compensate for the inherent weaknesses in language-supplied I/O processing may improve program portability.

b. Greater Flexiblity of Data Presentation

Increased flexibility in the presentation of output data is one of the major objectives behind this thesis. Allowing ready access to sophisticated I/O primitives gives the application programmer the power to adapt data presentation to fit the existing environment thus enabling him/her to take advantage of technological advances in interactive display techniques. In fact, it may be conceivable to allow the resident OS to make the necessary decisions involving display technique; additionally, it may be possible to give the user complete control of data presentation to fit his/her own needs or preferences.

c. Faster I/O Processing

For all but the most trivial I/O requests, processing may be significantly faster since drivers could be written to take advantage of specific hardware characteristics.

d. Ease of Concurrent Program Data Exchange

The exchange of data between concurrent processes may be greatly enhanced since an intermediate structure and a standard protocol will be available to facilitate data exchanges.

Although the issues discussed above may represent only a few of the possible considerations surrounding the interfacing dilemma, a thorough analysis can not be completed until actual implementation of the proposed interface has been completed.

III. DESIGN OBJECTIVES

A. PRIMARY DESIGN OBJECTIVES

From the previous sections of this thesis, two overall design objectives become apparent in the design of the interface: 1) a standard protocol for communications between application programs and the host system or between application programs themselves must be established, and 2) a consistent, flexible and simple interface mechanism has to be designed which can meet not only the diverse needs of the application programmer but also accommodate technological advances both in hardware and software.

A major obstacle which has inhibited proposals for development of an operating system interface has been the lack of established parameter passing conventions between high level application languages and low level system service drivers. Additionally, existing difficulties have been greatly compounded by the general unwillingness on behalf of a sizeable minority of application language implementors to comply with recognized standards for the internal representation of data as delineated by the IEEE [Ref. 1]. These differences in parameter passing conventions and internal data representation have contributed in a limited degree to software incompatability problems and in a larger capacity to the lack of software portability.

In light of these obstacles, the third and most important objective must be the design of a flexible mechanism through which a varying number of mixed application language typed variables may be translated to standard internal formats and passed as parameters to low level system service drivers.

B. ANCILLARY DESIGN OBJECTIVES

1. Maintainability and Extensibility

In order to achieve the second primary objective the interface design must be such that additions and changes to the existing interface can be made without destroying the integrity or the stability of its structure. In other words, the interface must be both maintainable and extensible yet, at the same time, remain invariant in its overall structure. Both of these objectives can be met through the design of an interface framework containing a substructure in which an unlimited number of loosely coupled modules may reside. Inclusion of such a substructure would permit the individual modules to be inserted, revised and deleted as necessary.

2. Accessability and Efficiency

To be of any practical use, the primitives must be easy to use and must take maximum advantage of the inherent hardware characteristics. That is, the interface must be efficient and easily accessed. Designing a mechanism that is easy to use means that the conceptual nature of the interface must be kept both simple and consistent throughout its overall design. Achieving maximum efficiency can be realized by ensuring that implementation of the primitives is totally transparent to the application program, thereby permitting technological updates without affecting program design (information hiding).

3. Transportability and Flexibility

Although source code transportability is primarily a language design issue, through the establishment of a universal communications protocol and a means of providing external I/O enhancements, the temptation on the part of language implementors to add nonstandard items to the host language will be reduced. This, in turn, would permit programmers to keep their program's main logic transportable and also permit greater flexibility in formatting program output.

4. Implementation Simplicity

To encourage immediate acceptance and use of the interface, simplicity of implementation is imperative. This implies that the proposed framework must fit readily into existing operating systems with a minimal amount of effort. Taking advantage of the more common primitives provided would be a viable approach to this end.

IV. PROPOSED INTERFACE DESIGN CHARACTERISTICS

A. THE 'DYNAMIC KERNEL' CONCEPT

To date, the emphasis towards standardization of microcomputer operating systems has revolved, primarily, around establishing a static set of basic primitives (a kernel) to be made available for use by programmers. Industrial standards have been slow to emerge because system software experts have failed, in the past, to acknowledge the increased demands by application programmers for more sophisticated and accessible system services. Recently the IEEE, in an effort to promote program portability, proposed a set of primitives to be included within the kernel of all operating systems [Ref. 2]. This was a significant step towards improving portability; however, the necessary mechanisms for accessing the proposed primitives from within high level languages and the intended strategy for future kernel revision were not substantially delineated.

Establishment of a universal static kernel for microcomputer operating systems, or for mini or mainframes for that matter, should be considered impractical, narrow in scope, and counterproductive due to its limited capacity to incorporate the rapid advancements in both hardware and software technologies. The emphasis toward standardizizion should focus instead upon the establishment of a universal protocol for data exchange between application programs and the host system and upon development of an extensible, flexible and uniform structure for embedding both primitive and high level system services within a 'Dynamic Kernel'. The remaining sections of this chapter describe a conceptual model which may conceivably be adopted as a standardized interface structure for incorporation of a 'Dynamic Kernel'. Successive chapters will address recommended primitives to be initially placed within the 'Dynamic Kernel' and possible implementation techniques.

B. A CONCEPTUAL OVERVIEW OF THE PROPOSED INTERFACE

1. Introduction

A brief overview of the major interface components and a simple example of how a basic system service request is processed are useful for understanding the conceptual nature of the proposed interface. More detailed descriptions of the individual interface components, component interactions, and service request processing are presented in the sections following the conceptual overview.

It must be empahsized that the descriptions which follow are intended to convey the conceptual aspects of the interface structure. Specific data structures and boundary values have been chosen only to demonstrate implementation feasibility. Actual implementation of the interface structure is by no means restricted to these choices and, in reality, during latter stages of implementation, it will more than likely be necessary to select data structures and boundary values which enhance efficiency. It is also reasonable to expect that several of the conceptual components of the interface would require integration into single multi-function modules, in order for the interface to operate within existing 'real world' memory constraints.

2. Overview

The interface structure consists of several separate but highly coupled components. The primary component of the interface can be viewed as a large, two dimensional array, residing in main memory, representing a general directory of generically grouped system services (e.g file management, video display functions etc.). Each element of the array, defined by the intersection of a row and column, can be imagined to contain a pointer to a dense index of related system services (Figure 4.1). This dense index (a three dimensional array), in a similar manner, contains elements holding pointers to the location of direct primitive calls, device drivers or sophisticated run time routines appended to the operating system. Interface drivers, necessary to initiate service requests and pass associated function parameters are linked into the application language source code. Data exchange between the application program and the service drivers takes place in areas, created dynamically in main memory, specifically allocated for this purpose.

For example, suppose an application program wished to delete a file residing in a secondary storage device. Assume that the element (1,3) in the main directory (the resident two dimensional array) holds a pointer to an index of all file management routines. Also consider that the element (in the index) described by the coordinates (5,3,2) holds a pointer to the location of the code segment which will fulfill the request. Then the two coordinate pairs ((1,3),(5,3,2)) would provide the application program direct access to the desired code segment. The code would then be executed with the exchange of necessary parameter and error condition information taking place in a data block which had been previously created dynamically and initialized prior to the request.

3. Component Definitions

With a broad understanding of the conceptual nature of the interface in mind, and unobscured by details, it is useful to assign descriptive names to several of the components of the interface in order to clarify future references.

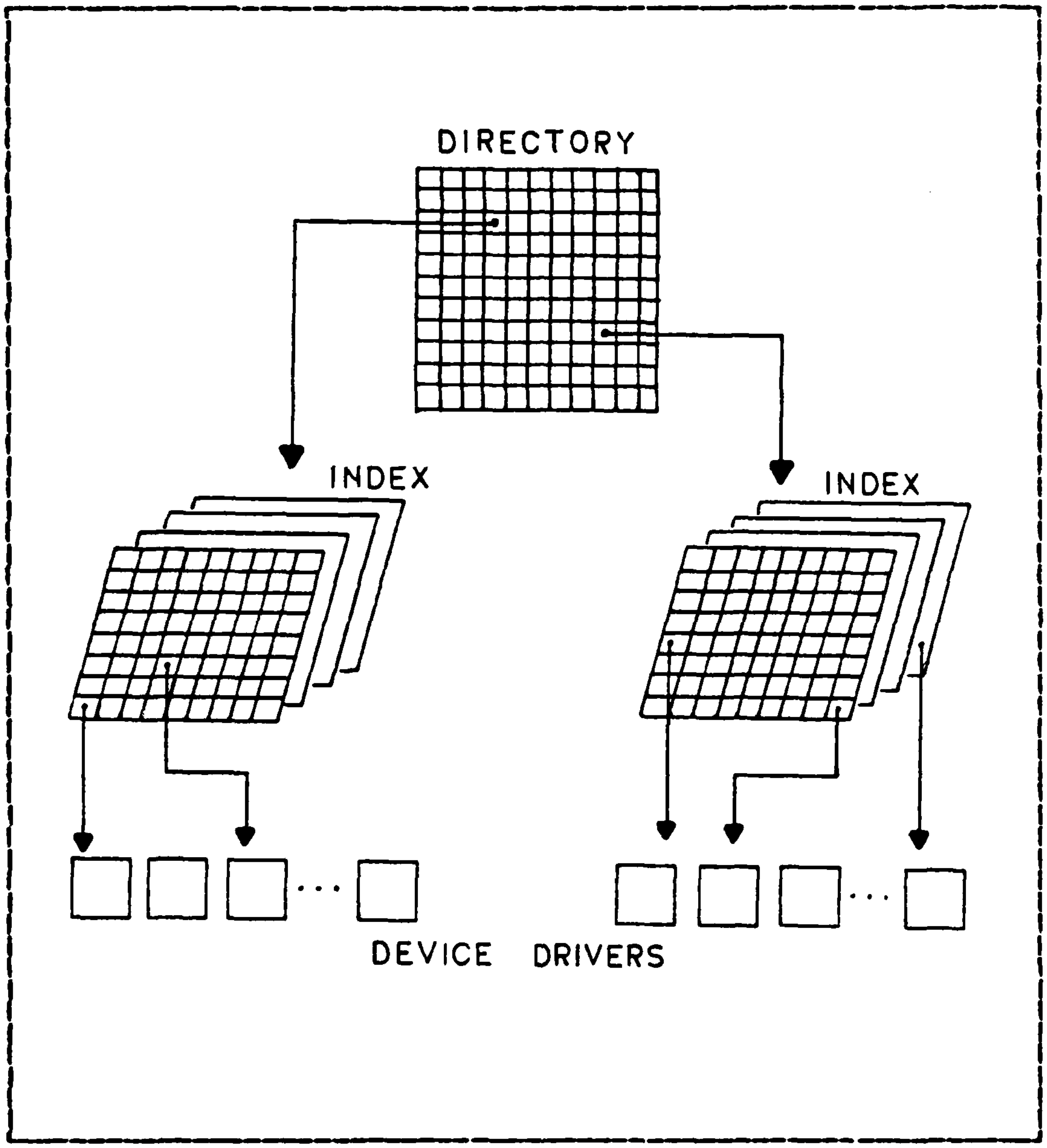


Figure 4.1 Conceptual View of Interface.

By its very nature the resident two dimensional array which functions as a directory to generic groupings of system services can be appropriately named the System Services Directory (SSD). In a similar fashion, the more detailed indexes (three dimensional arrays viewed as multi-paged volumes) which contain information for accessing specific and related system services will be referred to as System Services Indexes (SSIs). The dynamically created memory blocks used for exchanging parameter data will be referred to as Data Exchange Blocks (DEBs).

Several other components necessary to complete the interface are the Application Language Interface (ALI), the Index Paging Area (IPA), the Boot Time Processor (BTP), the Service Request Manager (SRM), the Data Block Manager (DBM) and the Service Drivers (SDs). These components, although essential for operation of the interface, were intentionally omitted from the brief overview above in order to ensure the basic conceptual mechanisms of the interface were not obscured by details.

C. INTERFACE COMPONENT DESCRIPTIONS

1. System Services Directory (SSD)

The System Services Directory (SSD) can be viewed as a two dimensional array, residing in main memory, that represents a general directory of all unrelated system services (e.g., file management, video display functions, etc.). Each element of the array, by virtue of its coordinate address, can be imagined to be an implied pointer to a dense index of related system services (Figure 4.2).

These elements actually contain two status bits which are vital to the operation of the interface. The first bit is used to indicate whether the selected generic category of system services has been implemented in the operating system interface. The second bit is used in conjunction with a page number to determine whether the requested index page is resident in the IPA (Figure 4.3).

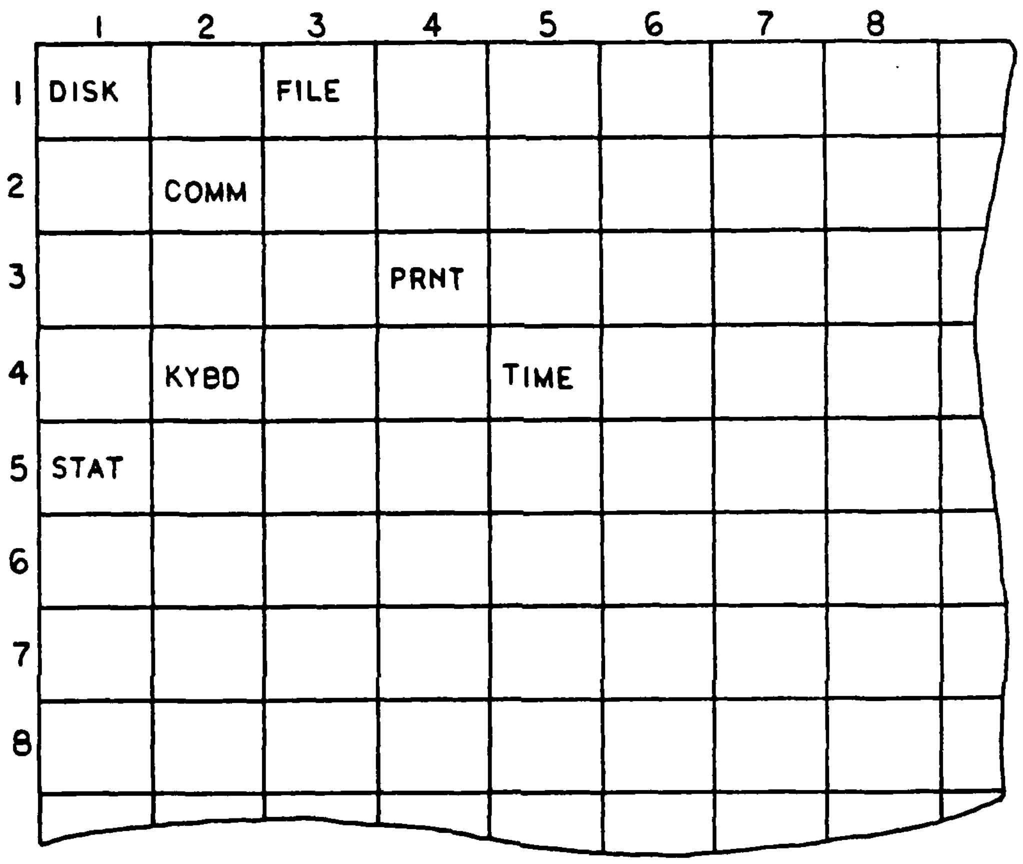
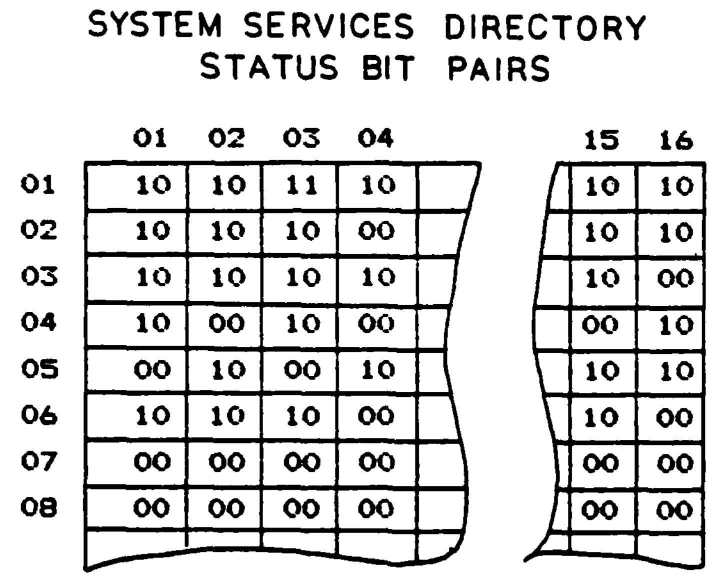


Figure 4.2 Conceptual View of SSD.

The actual size of the System Services Directory is not necessarily bounded; however, for practicality during implementation it is desirable to restrict its physical size such that it may fit within a reasonable area of memory in



SYSTEM SERVICES DIRECTORY
STATUS BIT PAIRS

	01	02	03	04	15	16
01	10	10	11	10	10	10
02	10	10	10	00	10	10
03	10	10	10	10	10	00
04	10	00	10	00	00	10
05	00	10	00	10	10	10
06	10	10	10	00	10	00
07	00	00	00	00	00	00
08	00	00	00	00	00	00

The bit pairs above may be interpereted to mean:

00 –	The generic services grouping is not installed in the SSD.
10 –	The generic services grouping is installed int the SSD.
11 –	The generic services grouping is installed in the SSD and a page of the Services Index is resident in the IPA.

Figure 4.3 Memory Image of SSD.

current microcomputers systems. If the size of the SSD is restricted such that it contains only 256 elements then the total memory required to be allocatad in main memory can be calculated as follows:

(256 elements) * (2 bits per element) = 512 bits

(512 bits) / (8 bits per byte) = 64 bytes

The 256 SSD generic groupings have an endless variety of possibilities, and, if the BIOS and DOS routines contained in existing operating systems are analyzed (see Appendices), it is evident that several generic groupings occur naturally. Among the most common are:

1. Direct disk access functions
2. Communications functions
3. Keyboard functions
4. Printer functions
5. System status requests
6. Video display functions
7. File management functions
8. System timer functions
9. Memory management functions

In addition to these commonly found groupings, it is not difficult to envision construction of other possible generic sets, such as:

10. Graphics function requests
11. Data encryption requests
12. User defined Macro definition requests
13. Database function requests
14. Output data formatting requests
15. Input data formatting requests
16. Data exchange requests (pipelines)
17. System utility requests (filters)

The generic groupings above represent only a small number of the total 256 directory entries and serve to illustrate that the capacity of the interface to accommodate numerous additions within the SSD is not significantly affected by the size limitations which have been imposed.

2. System Service Indexes (SSIs)

The System Service Indexes (SSIs) can be described as multi-paged two dimensional arrays whose elements point to the location of direct primitive calls, device drivers or high level run time services appended to the operating system. The addresses contained in the SSIs may point to actual memory locations, where a particular System Service Driver resides, or indicate that the selected Services Driver either has not been implemented or that it resides in an external file.

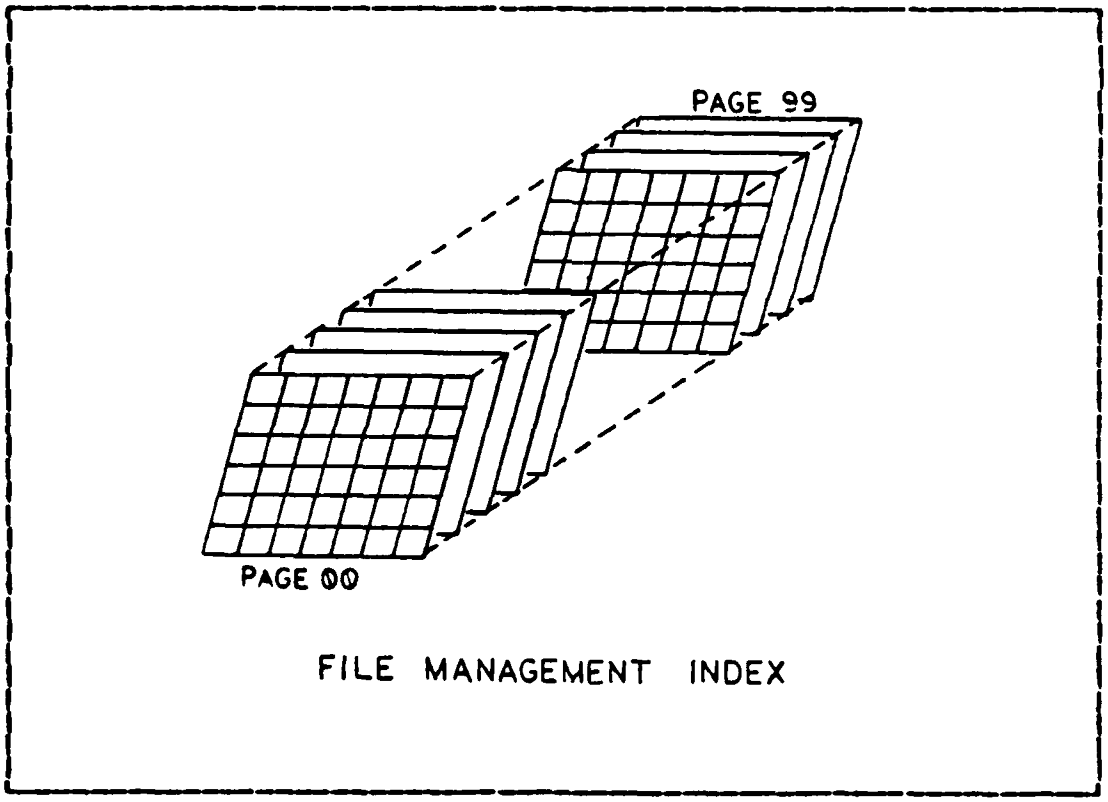


Figure 4.4 Conceptual View of System Service Index.

Each three dimensional SSI can be more illustratively envisioned as a single index volume containing many two dimensional index pages (Figure 4.4). The entries found on the index pages point to the location of independent code segments required to perform a specific task. These entries hold a single address which provides direct access to Service Driver code segments residing permanently in main memory or serve as a flag to indicate either non-implementation status or to indicate that the desired code segment resides in an external file (Figure 4.5).

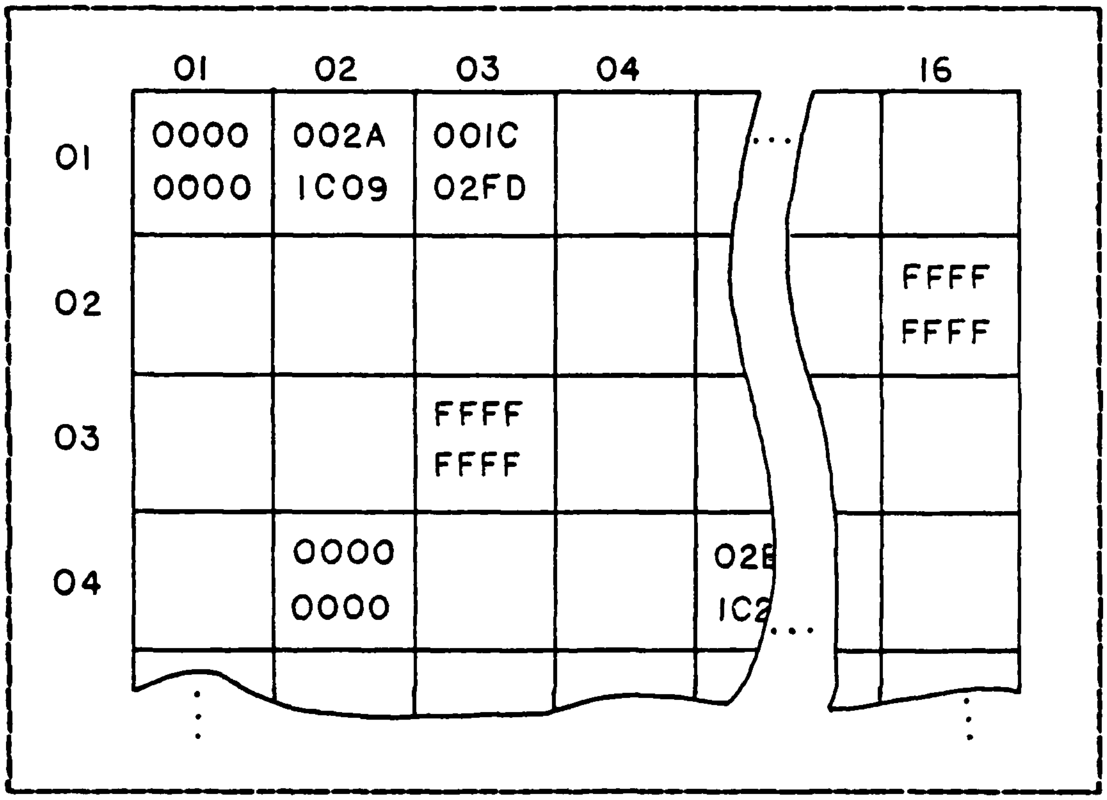


Figure 4.5 SSI Page for a Segmented Address Machine.

All SSI pages of the same page number are grouped together in a single random access file containing 256 records (one record for each of the 16x16 generic groupings specified in the SSD). Each index record within this file possesses 257 individual fields, with one field holding a generic type identifier and 256 others holding the address used to indentify where individual Service Driver code segments are located (Figures 4.6 and 4.7).

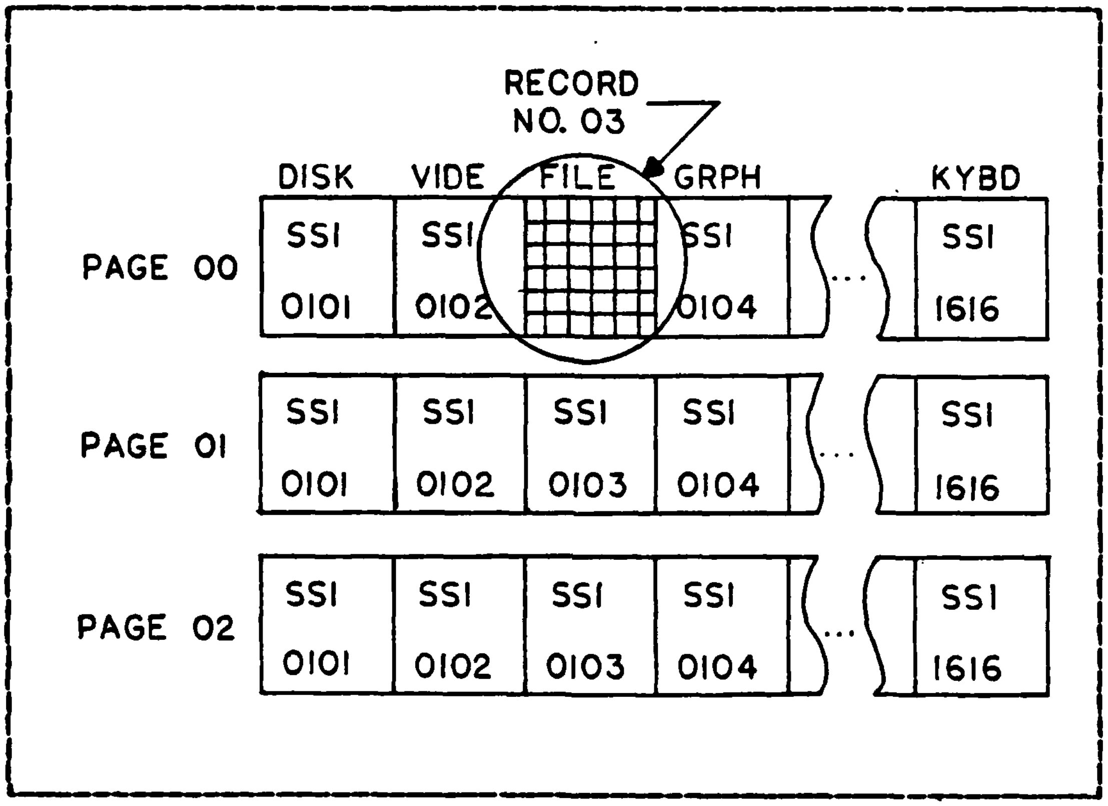


Figure 4.6 SSI Page Files.

If it is assumed that the imaginary target machine, for which the conceptual model was designed, is a 16 bit machine with a 20 bit address bus then each address field must be 32 bits in length. The first 16 bits of the field can therefore be interpreted as a segment address and the remaining 16 bits interpreted as a segment offset. Based on this assumption it would then be possible to indicate that a particular Service Driver has not been implemented by setting both 16 bit values to 0000h. Similarily, to indicate that a driver is stored in an external file (nonresident) the two 16 bit addresses may each be set to FFFFh.

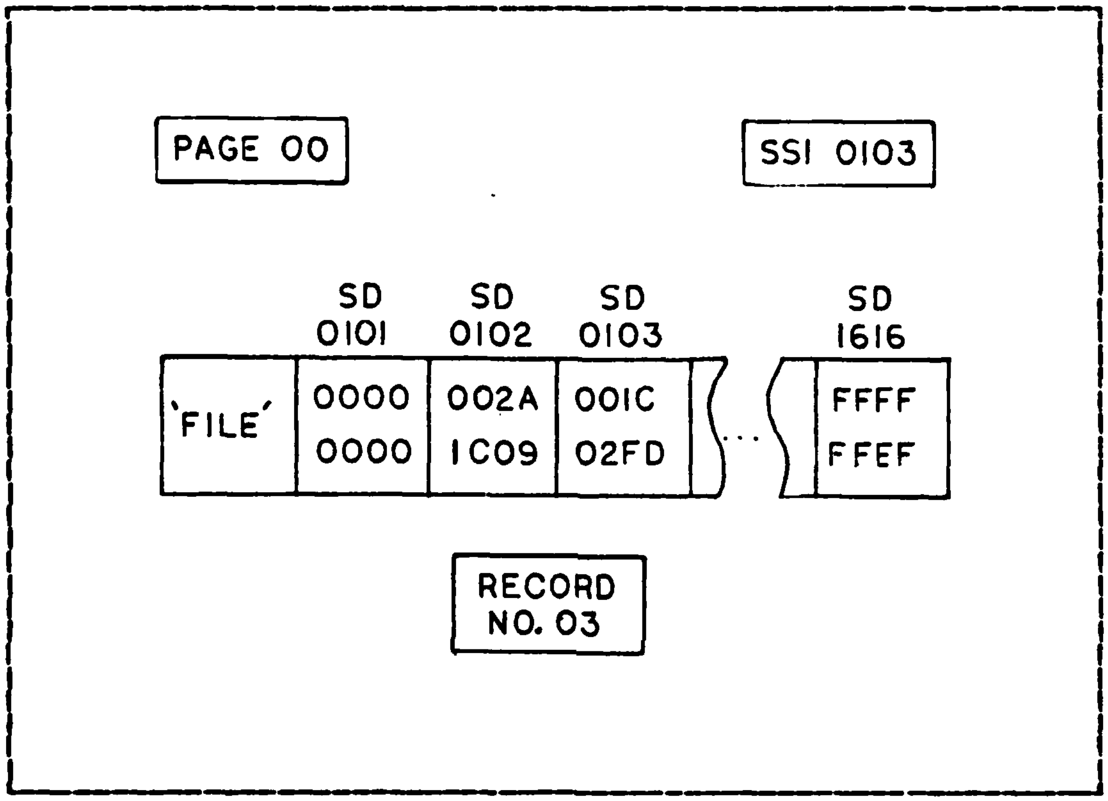


Figure 4.7 Field Contents of a SSI Page Record.

Retrieval of the correct Index record from the SSI Page file may be accomplished using the row and column numbers of the System Services Directory to calculate the proper record. (This can be accomplished by applying Equation 4.1.) Several random access blocks read may then be used to place the specific SSI page in the Index Paging Area (IPA).

Record Number = ((Row - 1) * 16) + Column (Eqn 4.1)

Once the correct record has been retrieved and placed in the IPA the desired Service Driver code segment may then be located. Should it be necessary to dynamically link an external code segment, the file containing the code may be located by appending the row and column numbers, used to initially locate the service in the Service Index, to the four letter generic type identifier, contained in the first field of the Service Index page record. To clarify this procedure through an example it will be assumed that the desired service is a video function (type identifier = VIDO) and that the service desired is located on page 04 (now resident in the IPA), row 03 and column 14 of the System Service Index. The external file which must be retrieved would therefore be VIDO0314.P04.

3. <u>Index Paging Area</u> (<u>IPA</u>)

The Index Paging Area (IPA) is a fixed area reserved in main memory which is used to hold transient System Service Index (SSI) pages. In its simplest form it may be viewed as a single two dimensional array whose size corresponds exactly to that of a single SSI page and in which only one Index page may be placed after a System Service Request has been generated via the System Service Manager. A more useful form (although much more complex) would be one which contained sufficient space to hold four Index pages. This would permit two Index pages to be used as primary defaults and provides space in order that two pages may be swapped in and out of memory on a demand or selection basis.

The memory which must be allocated for the IPA can be calculated by using Equations 4.2 and 4.3.

Page Size = (Rows * Cols.) * (Bytes/Element) (Eqn 4.2)

IPA Size = ((Page Size) * (No. Pagess)) + 4 (Eqn 4.3)

Using these formulas, the smallest IPA Size possible on the imaginary 16 bit target machine would then be:

Page Size = (16 * 16) * (4) = 1024 bytes (Eqn 4.4)

IPA Size = ((1024) * (1)) + 4 = 1028 bytes (Eqn 4.5)

The numerical calculation above clearly shows that a four page IPA could easily fit into the main memory of present advanced microcomputer systems. Considering that the majority of the newer personal computers are now being sold with an addressable memory space of 128K (or greater) the 4K required by the IPA is a relatively small sacrifice in terms of the net gain achieved by the interface.

4. Service Drivers (SDs)

Service Drivers (SDs) are code segments that perform the actual system service requested. It would be desirable, of course, for the more frequently used Service Drivers to reside in main memory (ROM/RAM); however, due to memory limitations, the vast majority would require storage on external devices (i.e., disk drives, tape drives, etc.).

The drivers may be written and provided by operating system vendors, equipment manufacturers, independent software houses or by application programmers. Whatever the source of the SD, it must be installed in the appropriate System Services Index and be capable of dynamic linking if it is to reside on secondary storage.

Each Driver looks for its necessary parameters in the DEB that is constructed to exact specifications delineated in the documentation provided with each Service Driver.

5. Data Exchange Blocks (DEBs)

Data Exchange Blocks (DEBs) are used for exchanging data between application programs and Service Drivers (SDs).

The incorporation of the Data Exchange Blocks into the interface is necessary due to the lack of a standardized parameter passing convention and the unwillingness of language implementors to adhere to the standard guidelines set forth by the IEEE for internal representation of data within computer hardware [Ref. 1]. These blocks can be best described as linear linked lists, dynamically created in memory, which serve as variable type conversion tables (Figure 4.8).

Data Blocks are created by the programmer via the Data Block Manager in order to provide a direct path for communications between the application program and a single Service Driver or a number of closely related Service Drivers. The particular format of individual Data Exchange Blocks is directly related to the parameter passing conventions of the Service Drivers; these conventions are explicitly delineated in the documentation.

Every DEB consists of a header record and additional records whose number and *specific* order is dictated by the Service Driver documentation. The header contains a current count of the records in the list and a pointer to the first record. Each remaining record contains a pointer to the location in memory of an application language variable, an application language variable typing descriptor, a standardardized variable typing descriptor and a pointer to the location in memory of a temporary variable. The exact nature and purpose of the variable typing descriptors and the temporary variables will be addressed in the next section which deals with the Application Language Interface (ALI).

6. <u>Application Language Interface</u> (<u>ALI</u>)

The Application Language Interface (ALI) is a collection of run time routines which performs two way

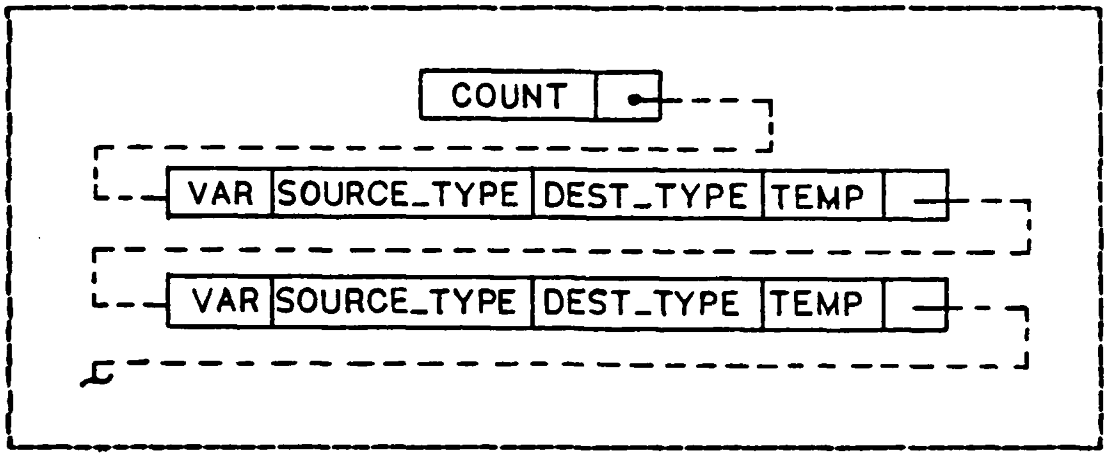


Figure 4.8 Conceptual View of Data Exchange Block.

translations of application language typed variables and standardized typed variables in order to provide two way communication between the Service Request Manager and Service Drivers or between concurrent application programs. This translation is necessary due to the differing variable formats used by high level languages despite recommended standards. As a result the ALI, by necessity, is extremely language dependent and therefore must be implemented with a particular target application language in mind. The standardized typed variables used during the translation process are assumed to be those which have been adopted as a standard for internal machine representation by the IEEE [Ref. 1].

A calling module requests translaton services by passing the address of a Data Exchange Block to the ALI in addition to setting a boolean switch to indicate in which direction the translation is to take place. For a forward translation request the ALI locates the application language variables, performs the required translations (based on the information provided by the typing descriptors) and then places the translated variables in temporary locations. The

ALI then places the addresses of the temporary variables in the Data Exchange Block and finally returns control to the calling module. A reverse translation is performed in much the same manner using the application language variables and standardized temporary variables in reversed roles.

Obviously, compliance with established standards for internal data representation would make the data format translation process unnecessary. The result of universal adherance to these standards would not only markedly increase the overall performance effeciency of the interface but would also significantly reduce its complexity.

7. Data Block Manager (DBM)

The Data Block Manager (DBM) is an external code segment that is linked into the program source code. The DBM enables the programmer to create or destroy Data Exchange Blocks as well as append or remove entries within individual Data Blocks in order to meet Service Driver specifications.

The programmer communicates with the Data Block Manager via a Data Block Interface (DBI) whose format is shown below along with several examples to illustrate its use.

```
BLOCK(OPERATION,BLK_ID,VAR,SOURCE_TYPE,DEST_TYPE)
```

where:

OPERATION = A decimal integer used to indicate one of four operations to be performed on a block referenced by BLK_ID:

1 - Create a new block
2 - Destroy an existing block
3 - Add a variable to a block
4 - Remove a variable from a block

BLK_ID =	A pointer within the source language to a specific Data Exchange Block.
VAR =	The address of a variable defined in the source language which is used to exchange data between the application program and a particular Service Driver.
SOURCE_TYPE =	A decimal integer used to identify source language variable typing. The appropriate typing code must be obtained from documentation furnished with the ALI. (Value Range: 00 - 99)
DEST_TYPE =	A decimal integer used to specify standardized variable typing. The appropriate typing code must be obtained from documentation furnished with the ALI. (Value Range: 00 - 99)

8. Service Request Manager (SRM)

The Service Request Manager (SRM) is an external code segment which must be linked into the application language source code and is responsible for initiating and performing system service requests by application programs. Requests for system services are made via the Service Request Interface (SRI) by the applicaton program and the requested services are provided by the SRM through execution of appropriate Service Driver code segments.

A more detailed description of the Service Request Interface format (from the viewpoint of an application programmer) is shown below.

```
SYS_REQUEST(SSD,SSI,PAGE,ERROR,BLK_ID)
```

where:

SSD = A decimal integer representing the coordinates of a specific generic catagory of system services described in the System Services Directory. The first two digits represent the row number and the last two digits represent the column number.
(Value Ranges: 01|01 - 16|16)

SSI = A decimal integer representing the coordinates of the required Service Driver described on the selected System Services Index page. The first two digits represent the row number and the last two digits represent the column number.
(Value Ranges: 01|01 - 16|16)

PAGE = A decimal integer identifying a specific index page of the selected System Services Index.
(Value Ranges: 00-99)

ERROR= A decimal integer returned to the application program by a Service Driver in order to describe specific error conditions which have occurred, thus permitting graceful recovery from error conditions.
(Value Range: 00000 - 99999)

BLK_ID =A pointer to a Data Exchange Block which has been formatted for use by the selected Service Driver.

After a request for services has been generated the Service Request Manager is responsible for checking SSD, SSI, Page and Error range values. Additionally it must request services from the Application Language Interface (ALI) prior to passing the selected Service Driver the address of the identified Data Exchange Block (DEB). Once these steps have been completed it must then access the driver code segment, pass the address of the DEB to the driver, execute the driver code and finally return any error condition status codes via the global Error variable.

9. Boot Time Processor (BTP)

The Boot Time Processor (BTP) is responsible for allocating memory for the Index Paging Area (IPA) and the System Services Directory (SSD) as well as ensuring that the SSD is initialized to reflect which generic groups have been installed. Beyond this it serves no other purpose and is considered a separate component of the interface merely to complete the interface design.

10. Examples of Interface Processing

The following example and accompanying illustrations below are intended to demonstrate how a typical source language program service request may appear and clarify the overall intercomponent relationships within the interface.

The example assumes that the System Service Requested is to scroll the video display 10 lines within a specified rectangle on a standard CRT. Additionally, the documentation provided with the Service Driver indicates that the driver expects to find the addresses of five integer values in the following order:

1. Lines:INTEGER = number of lines to scroll
2. X1:INTEGER = x coordinate of upper left rectangle corner
3. Y1:INTEGER = y coordinate of upper left rectangle corner
4. X2:INTEGER = x coordinate of lower

right rectangle corner

5. Y2:INTEGER = y coordinate of lower
   right rectangle corner

A typical program source code segment may appear as:

```
(* DECLARE VARIABLES *)

Scroll_Data : POINTER;
N_lines,Upper_x,Upper_y,Lower_x,Lower_y : INTEGER;
Append,Create,Video,Scroll,Page,Error: INTEGER;


(* ASSIGN SSD AND SSI COORDINATES *)
(* AND DEFINE BLOCK OPERATIONS.   *)

Video:=0302;    (* SSD GENERIC CLASSIFICATION *)
Scroll:=0508;   (* SSI COORDINATES OF DRIVER *)
Page:=3;        (* PAGE NUMBER OF SSI *)
Create=1;
Append=3;


(*****   MAIN PROGRAM CODE   *****)

(* CREATE AND FORMAT DATE EXCHANGE BLOCKS *)

BLOCK(CREATE,Scroll_Data,0,0,);
BLOCK(APPEND,Scroll_Data,N_lines,1,1);
BLOCK(APPEND,Scroll_Data,Upper_x,1,1);
BLOCK(APPEND,Scroll_Data,Upper_y,1,1);
BLOCK(APPEND,Scroll_Data,Lower_x,1,1);
BLOCK(APPEND,Scroll_Data,Lower_y,1,1);

(*  SPECIFY REQUEST PARAMETERS  *)

N_lines:=10;
Upper_x:=5;
```

```
  Upper_y:=5;
  Lower_x:=20;
  Lower_y:=60;
  Error:=0;

  (* INITIATE SERVICE REQUEST *)

  SYS_REQUEST(Video,Scroll,Page,Error,Scroll_Data)

  .            .              .                .
  .            .              .                .
  .            .              .                .

(*******  END MAIN PROGRAM  *******)
```

11. Remarks

Once again it should be emphasized that the data structures and boundary values used in this conceptual description of the interface by no means confines future implementation schemas. A variety of methods may be used to achieve the same end; however, the important point to recognize is that, to the application programmer, the actual physical implementation of the interface must be totally transparent and that efficiency of operation is of the utmost importance.

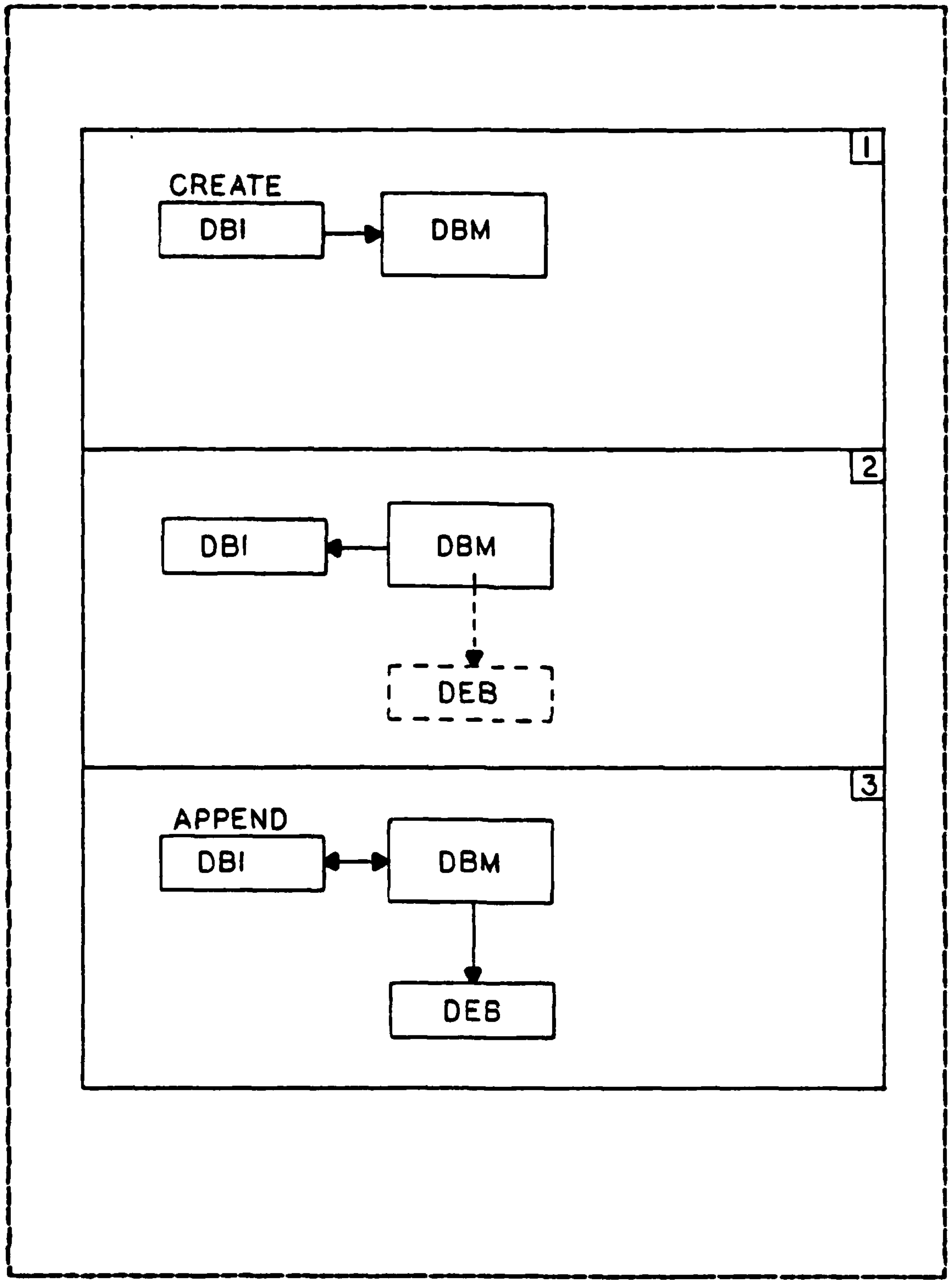


Figure 4.9 The Service Request Process.

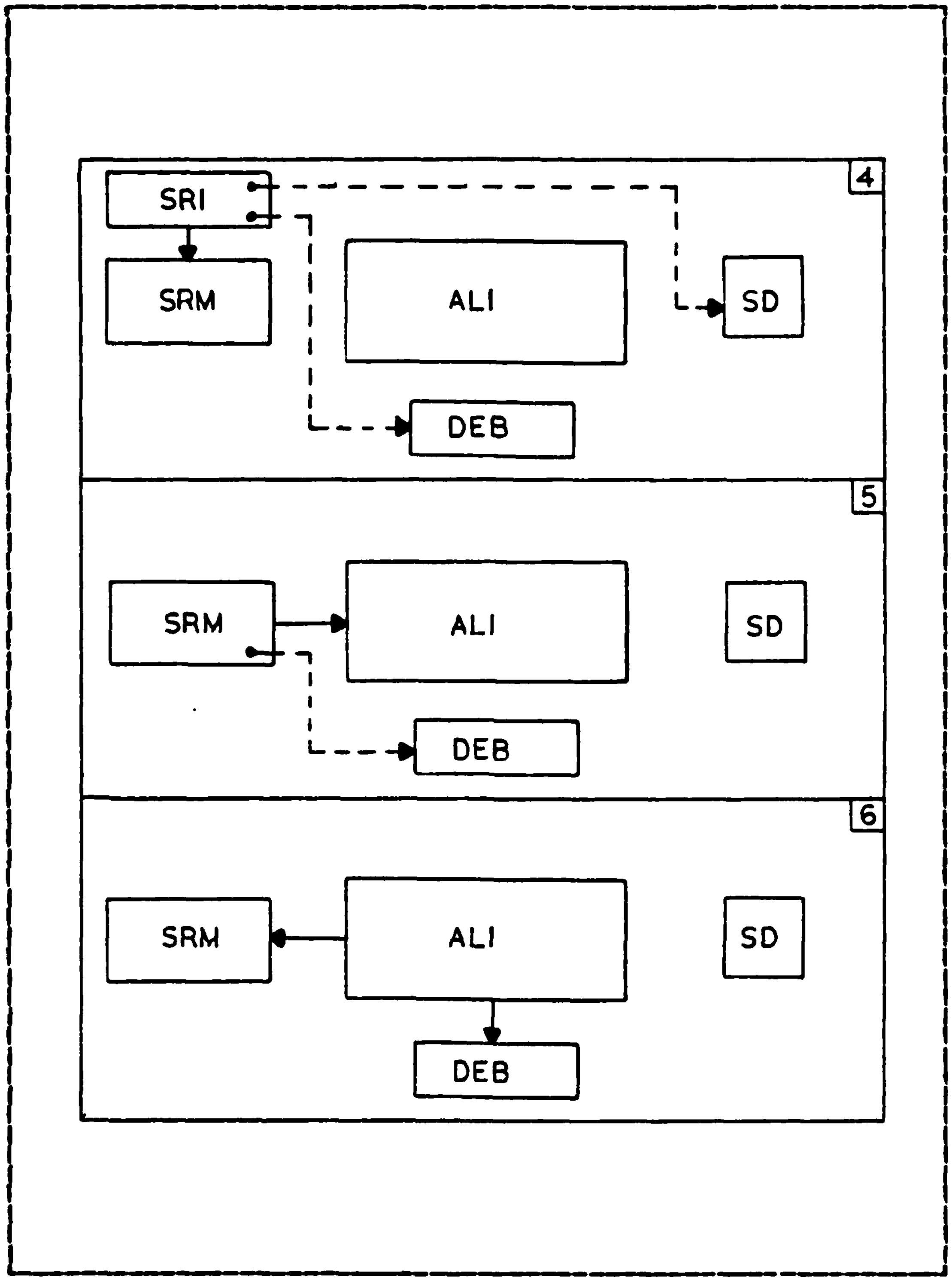


Figure 4.10 The Service Request Process (Continued).

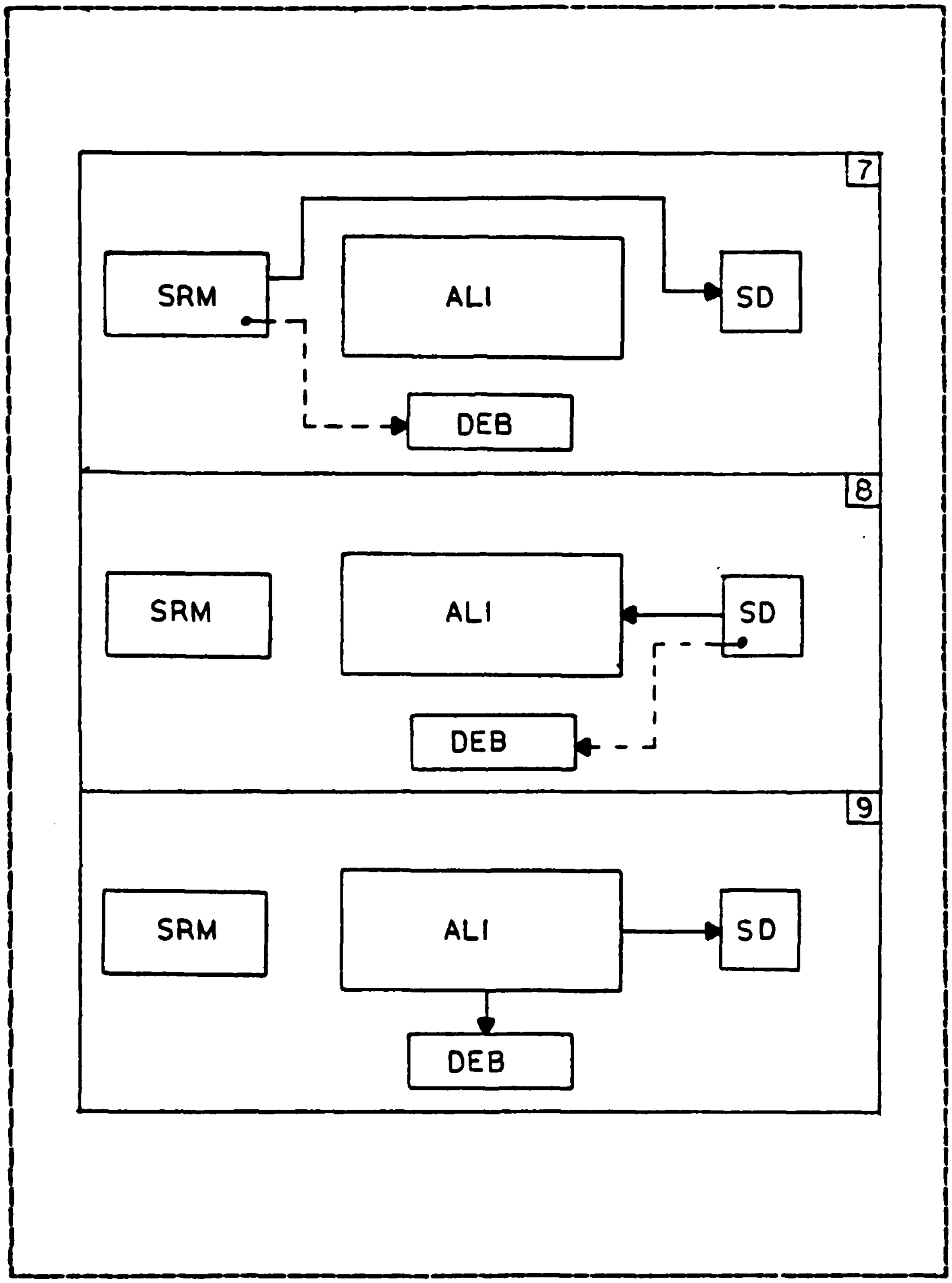


Figure 4.11 The Service Request Process (Continued).

V. PROPOSALS FOR STANDARDIZATION OF INTERFACE

A. STANDARDIZATION METHODOLOGY

It should be obvious to the reader at this point that the proposed interface can effectively resolve the issue of the establishment of a standard protocol for communications between application programs and the host system as well as between application programs themselves. Perhaps what may not be so obvious, however, is that the interface serves as a mechanism to achieve not merely a static set of primitives but, rather, a means for the creation of a 'Dynamic Kernel' which can be adjusted to meet future demands of application programmers as well as to accommodate the rapid changes in hardware/software technology.

This 'Dynamic Kernel' can be realized through partitioning the one hundred possible pages (levels) of the System Services Indexes into three distinct areas of authority. Each of which is reserved for use strictly by either standards reccommending bodies (e.g., ISO, IEEE, etc.), operating system designers (and equipment designers) or application programmers (Figure 5.1).

Creation of these partitions ensures a significant degree of flexibility and extensibility and at the same time provides a means of updating the 'Dynamic Kernel' (Level 1). As operating system utilities and other high level system services become increasingly more popular they may be standardized and placed within the 'Dynamic Kernel' by the governing establishment.

Several additional benefits may be realized by using this approach. First, it conceivably creates a broader base for the availability of systems software by encouraging

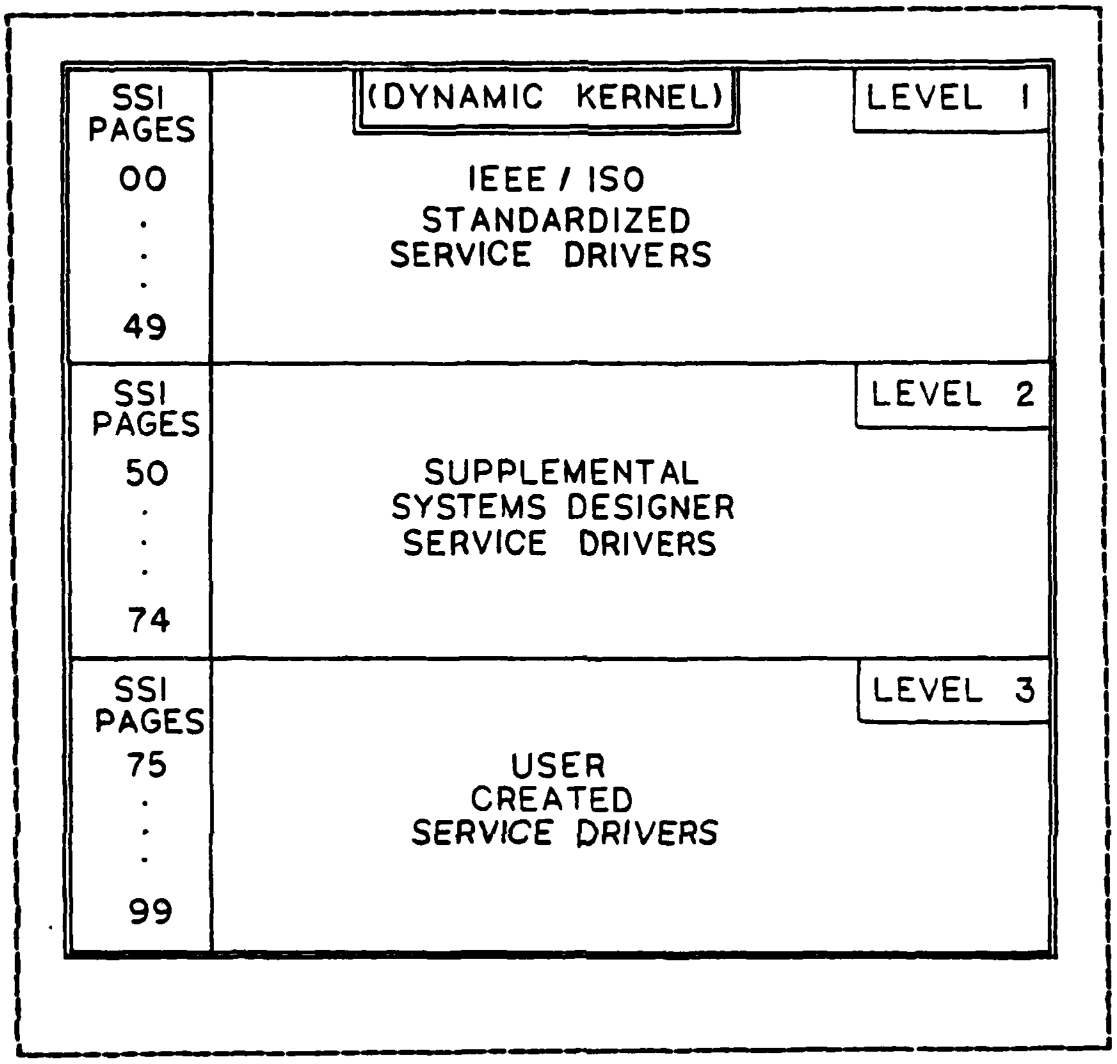


Figure 5.1 Example of Authority Level Partitioning.

independent software companies to allocate a portion of their development efforts towards generating new or upgraded modules for the two lower levels of the interface. Secondly, it permits the market place to have a direct voice in determining which utilities and services are to be selected for inclusion in the 'Dynamic Kernel'.

B. RECOMMENDED PRIMITIVES

As established in an earlier chapter, the vast majority of primitives furnished by existing microcomputer operating systems can be grouped into a limited number of broad catagories. These categories and readily available primitives can form the foundation for implementation within the prototype's 'Dynamic Kernel'. Listed below are the recommended generic groups and associated primitives within each group. The selected primitives below do not embody all those recommended by the IEEE [Ref. 2].

The selection of the primitives to be included in the prototype model was based on the services which are most readily available on popular 16 bit personal computers. Although several additional primitives have been included that are not generally available, their extreme usefulness and potentially simple implementation make them natural candidates for inclusion in the prototype's kernel.

1. Video Functions

a. Set video mode

Used to select desired video display mode (e.g., 80x25 text, 640x200 B/W graphics).

b. Set cursor position or advance cursor

Used to place the cursor at any position x,y on the video display (video mode dependent).

c. Read Cursor Position

Returns the present cursor position in terms of x,y coordinate values (video mode dependent).

d. Set Cursor Mode

Used to alter cursor display (invisible, half block, underscore, etc).

e. Select Active Page

Selects which of the multiple video display pages is currently being written to.

f. Set Display Page Size (Window)

Creates a window at x1,y1:x2,y2.

g. Scroll Displayed Page Up

Scrolls displayed page up n lines (scroll within established window only).

h. Scroll Displayed Page Down

Scrolls displayed page down n lines (scroll within established window only).

i. Clear Active Page

Clears active page (if displayed page is the current active page then clears within established window only).

j. Select Displayed Page (Swap in Block)

Uses block move to replace entire displayed page with page designated.

k. Read Pointing Device Position

Return x,y coordinates of point device position.

l. Read Character Attribute

Return byte(s) that provide(s) information about the present screen attributes of a character.

m. Write Character Attribute

Reset bit(s) that assign(s) screen attributes of a character.

n. Write Character at Location x,y

Write a character (or, if in graphics mode, a dot) at specified relative x,y position.

o. Write Character(s) at Cursor Position

Write a string of characters (or, if in graphics mode, a series of dots) at specified relative x,y position (truncate if it exceeds window boundary).

2. Direct Disk Functions

a. Reset Disk Drive System

Perform necessary buffer transfers to files, close all opened files and perform warm boot of disk drive system.

b. Return Disk Status

Return a byte(s) of information concerning the success or failure of file operations or mechanical malfunctions.

c. Read Disk Sector(s)

Perform absolute read of sector(s) specified.

d. Write Disk Sector(s)

Perform absolute write to sector(s) specified.

e. Verify Sector(s)

Verify sector(s) specified.

f. Format Track(s)

Format specified track(s).

g. Return Disk Type

Return information on current disk (e.g., number of tracks per inch, density, sector skewing, etc).

h. Set DTA

Set the absolute starting address of the Data Transfer Area to the specified address.

i. Return Buffer Count

Return the present number of buffers being used for file transfer in 128 byte increments.

j. Set Buffer Count

Set the present number of buffers being used for file transfer as specified.

3. <u>File Management</u>

a. Sequential Read

Perform sequential read of a specified file and place in the File Control Block(s).

b. Sequential Write

Perform sequential write of data in the File Control Block(s) to a specified file.

c. Random Read

Conduct random file read of record n.

d. Random Write

Conduct random file write to record n.

e. Return File Size

Return size of specified file to nearest 128th byte.

f. Return File Update Time

Return time that file was last updated.

g. Random Block Read

Conduct absolute block read at record n. Indicate if wrap around or partial read.

h. Random Block Write

Conduct absolute block write at record n. Indicate if insufficient space in record.

i. Parse Filename

Parse proposed file name to determine if valid format.

j. Return Current Path

Return current directory path in hierarchical directory.

k. Reset cCrrent Path

Reset current directory path in hierarchical directory.

l. Return Directory Count and Space Available

Return the number of entries present in the directory (or directory path) and return available disk space available.

m. Return Next Path Entry

Return the next entry in directory path.

n. Search Directory

Search fcr and remain at specified file in current directory path.

o. Create New File

Create new file in next available FCB.

p. Open Existing File

Find specified file in current directory and open file. Use next available FCB.

q. Close Open File

Close specified file and reset and return FCB as unused.

r. Set Open File Count

Reset the number of possible open files.

s. Copy File

Copy an entire file to specified destination.

t. Rename File

Rename indicated file to specified file name.

u. Return File Attributes

Return indicated file attributes (e.g., hidden, read only, etc.).

v. Set File Attributes

Reset specified file attributes in indicated file.

w. Delete File

Remove indicated file from directory and recover the space the previous file occupied.

4. Keyboard Functions

a. Toggle Control Break Enable

Enable or disable control-break key.

b. Toggle Escape Enable

Enable or disable escape key.

c. Return Alternate Keys Status

Return status of special purpose keys (e.g., CAPS key toggled, etc.).

d. Return Keyboard Character Code

Return scan code of key which has been pressed.

e. Flush Keyboard Buffer

Clear all characters from keyboard buffer.

f. Disable Keyboard Input

Disable the keyboard except for control-break and escape keys.

g. Assign Function Keys

Assign function keys a character string or new scan code.

h. Reassign Key Character Code

Reassign normal key a new scan code.

5. <u>Memory Management Functions</u>

a. Return Onboard Memory Count

Return system addressable memory installed.

b. Return Unused Memory Count

Return total unused addressable memory available.

c. Return Count Largest Block

Return size of largest unused memory block.

d. Set High Memory

Set highest program usable memory address.

e. Set Low Memory

Set lowest program usable memory address.

6. <u>Timer Functions</u>

a. Set Current Time

Set current time of day (24 hr) as indicated or from specified address.

b. Set Current Date

Set current date as indicated or from specified address.

c. Return Current Time

Return the current time of day.

d. Return Current Date

Return the date.

e. Set Timer On

Initialize and start timer.

f. Return Timer Status

Return timer count and start/stop status.

7. <u>Communications Functions</u>

a. Wait for Device Character

Wait for and return a character from external device unless time out reached (time out is specified in 10ths of a second).

b. Output Character to Device

Output character to external device.

c. Set Device Status

Set indicated device status byte(s) as indicated.

d. Return Device Status

Return indicated device status byte(s).

8. Printer Functions

a. Initialize Printer

Clear printer buffer and send reset signal.

b. Output Character

Output a character to printer.

c. Define Printer Table Code Sequence

Place a sequence of printer control characters in printer escape code definition table and assign indicated name to sequence.

d. Output Printer Table Code Sequence

Output named printer escape code sequence as defined in printer escape code sequence table.

e. Add to Print Queue

Add indicated file to print queue for printing.

f. Remove from Print Queue

Remove indicated file from print queue (stop print if in print).

g. Flush Print Queue

Clear entire print queue.

h. Return Current in Print

Return name of current file presently being printed.

i. Return Next in Queue

Return name of next file (from indicated position) in print queue.

9. <u>System Status</u>

a. Return System Service Implementation Status

Return code to indicate whether specified System Service Indexes or a particular Service Driver has been installed in the interface.

b. Reboot System (Cold Boot)

Perform hardware reboot of system.

c. Return Status Logical Units

Return code to indicate whether a specified logical unit is attached to system.

C. REMARKS

The selection of the primitives above was based on the author's personal biases and desire for ease of implementation. However, the Author also recognizes that should the interface model proposed in this thesis achieve general acceptance, the 'Dynamic Kernel' must be revised to conform to the IEEE standards [Ref. 2]. However, it is hoped that several of the additional primitives recommended above will be considered and approved for acceptance in the initial 'Dynamic Kernel'.

Regardless of the contents of the initial 'Dynamic Kernel', desirable services may be attached at one of the lower levels. Therefore, accessability to these services is always ensured; thus the interface will have fulfulled its purpose.

VI. CONCLUSIONS

A. SOME INTERFACE PROTOTYPE IMPLEMENTATION AND TESTING SUGGESTIONS

1. Target Machine

The suggested target machine chosen for development of the interface prototype is the IBM Personal Computer. It was selected primarily for the convenience of the author due to his familiarity with the machine and because a system was conveniently available in his home. A second reason for choosing the IBM-PC, which runs a version of MS DOS as the host operating system, is the growing popularity of sixteen bit machines in the market place. This does not preclude implementation of the prototype on eight or thirty-two bit machines, since one of the objectives in designing the interface was ease of implementation on all existing machines. Additionally, the growing popularity of MS DOS (a single user, nonconcurrent UNIX look-alike) makes it an ideal vehicle for broad-based analysis of the interface effectiveness.

2. Implementation Methodology

Actual implementation of the interface structure on the target machine should be able to be accomplished with only moderate effort. However, a sound top-down implementation strategy must be employed in order to permit use of a 'code then test' methodology during the implementation phase. This type of strategy is essential because it is assumed that only one individual will be involved in the process and a strategy of this type helps reduce overall debugging efforts and provides a natural approach to developing adequate documentation.

The accompanying hierarchical diagram (Figure 6.1) provides a quick pictoral review of the interface's conceptual structure and calling hierarchy. The initial prototype should use data structures and boundary values as close as possible to the conceptual interface model. This is suggested because it not only gives the implementation phase a clear and predetermined direction but also facilitates isolation of any conceptual and implementation level anomalies which may be discovered in the interface.

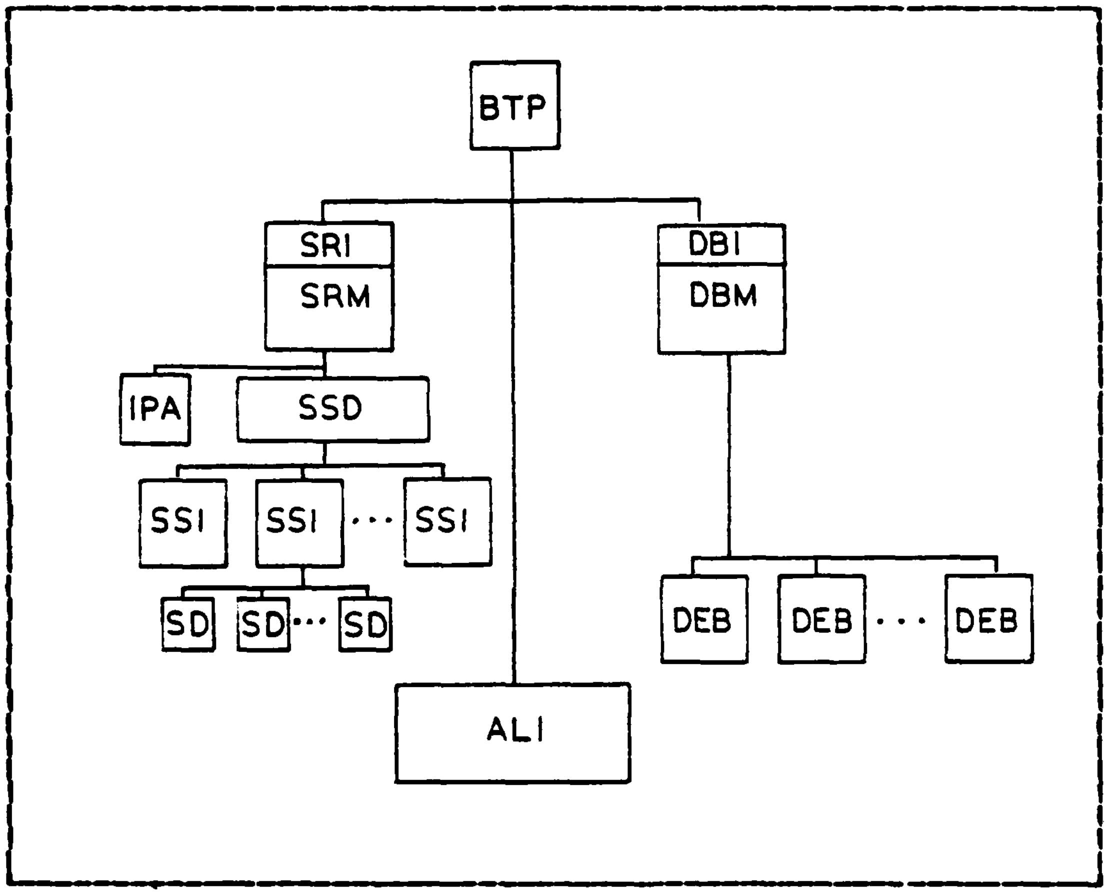


Figure 6.1 Interface Hierarchical Diagram.

3. General Suggestions

a. Select a Suitable Development Language

A suitable choice for the developmental language would be 'C' which was designed primarily as a system development tool and which is widely available for use on microcomputer systems. An additional motivation for choosing 'C' is the availability of numerous development tools found on larger UNIX based machines.

b. Create the System Services Directory in Memory

It is extremely tempting to place the System Services Directory (SSD) in the user defined area of the IBM-PC's interrupt table; however, this detracts from the portability of the interface. Creating the SSD in main memory is the only feasible method of implementation in most eight bit machines and a few sixteen bit machines. Additionally, creating the SSD in memory not only keeps the code necessary for the Index Paging Area extremely straight forward but also helps to reaffirm conceptual consistency.

c. Use a Single Page IPA

For simplicity during implementation, construction of a single page IPA in system memory is useful. However, a single page IPA restricts the choice of swapping methods to that of a pure 'demand' strategy. This strategy may significantly reduce the overall interface performance and eventually necessitate employing a multiple page IPA in order to assure a more reasonable performance evaluation.

d. Create a Single Multi-Purpose Service Driver Test Stub

A top-down design requires the use of numerous stubs; however, if these stubs are designed with care they serve as more than a mere vehicle for the 'code and test' strategy. They can in fact be designed to make coding of the module they are simulating an easier task when the appropriate time comes; in addition they can provide invaluable insight into potential logic problems. This is especially true in the case of the stub necessary for simulating the Service Driver.

The most obvious choice for this Service Driver test stub is one which permits access to the greatest number of system primitives available. In the case of the IBM-PC, which is interrupt driven, two assembly language programs provided especially for this purpose are included in the Norton Utilities Package. This package contains several useful software development utilities and is easily available for purchase from almost any major software distributor. The two most useful utilities provided in this package are designed to provide easy access to BIOS and DOS functions from within program source code. Linking this assembly language code segment to program object code is the same procedure required for attaching the Service Request Interface. Only slight modifications are necessary to combine these two utilities into a single code segement which may serve as the multi-function Service Driver test stub.

e. Create Generic Error Code Groupings

Each generic grouping within the SED should be assigned its own block of error code numbers. This helps to create a more logical and easy-to-use error code cross reference table that is grouped together by generic indexes and pages. A side benefit, of course, is that allocation of error codes in this fashion makes it easier to assign unambiguous error codes when attaching new Service Drivers.

B. EVALUATION

The evaluation phase should answer two general questions: 1) have the design objectives been met, and 2) does the interface effectively enhance software development for the microcomputer. For convenience the initial design objectives are summarized below.

A. Primary Design Objectives

1. A Standard Protocol for Communications
2. A Consistent and Simple Interface.

B. Ancillary Design Objectives

1. Maintainability and Extensibility
2. Accessability and Efficiency
3. Transportability and Flexibility
4. Implementation Simplicity

Reviewing these initial design objectives it is clear that the evaluation phase is a long term process and requires a broad base for testing. Therefore, any discussion of the evaluation process serves no practical purpose in this thesis other than to point out the major issues it must address.

C. POSSIBLE FUTURE WORK

1. Interface Enhancements

The following thoughts are provided for possible work beyond actual coding, testing and evaluation of the interface prototype.

a. System Service Index Installation Utilities

Utilities designed for the specific pupose of installing System Service Indexes and Service Drivers are essential tools which must be made available to interface users. These utilities should be simple to use, and should certainly provide extensive error checking and useful help facilities. The format used obviously must be highly interactive. Therefore, a menu driven format is a logical choice since this type of format helps reduce user input errors significantly. An informative and extensive on line help facility which is context sensitive is also an invaluable way to help reduce errors. As a matter of personal preference the author has found that help facililities which utilize graphical aids extensively tend to convey information much more efficiently than those which simply present textual definitions and procedures.

b. Documentation Utility

A very useful utility designed to scan source code for interface calls and to generate meaningful documentation of these calls within the source code most certainly would be a great step towards increased programmer productivity as well as code maintainability.

c. Incorporate Interface Components into the O/S Command Language

Access to the interface from within the host system's command language would provide the user with a very powerful shell development tool. Naturally, the services made available for execution in a direct mode should be carefully screened prior to making them available due to their possible destructive results. Yet there is little reason to place restrictions on commands issued from within command files (batch files). In some cases, the use of literals to refer to system requests is possible by utilizing the aliasing facilities found in many of the newer UNIX look-alike operating systems. This would permit the user to define his own command language that would fit his or her own personal needs.

d. Enhanced IPA Swapping

For the purpose of the prototype, it was assumed that the Index Paging Area (IPA) could only accomodate a single index page while utilizing a strict 'demand' swapping policy. In order to reduce table swapping, it would be beneficial to provide a larger four table IPA that could accomodate a primary and secondary default set of index pages as well as a two table area in which index pages are swapped using a 'frequency of demand' strategy coupled with a user directed default page definition.

e. Incorporate GKS and DES

One of the possibilities described earlier in this thesis was the inclusion of the Graphics Kernel Set (GKS) [Ref. 3] which has been considered as a graphics standard by the ISO. Additionaly, the growing popularity of electronic mail and rapid growth of telecommunicatons dictates the inevitable acceptance of a widespread public key encryption system. To that end inclusion of the Data Encryption Standard (DES) public key system [Ref. 4] in the 'Dynamic Kernel' is a logical enhancement to the prototype.

f. Attach Basic Database Management Services

This particular enhancement would be a major accomplishment itself because it would require very careful selection of the basic services to be provided. Furthermore, data format transparency consistent with the rest of the interface model is essential. Some of the basic services might be modeled after those found in Ashton-Tates 'dBase II' which is very popular in the personal computing community.

g. Provide Eloquent Data Formatting Services

The inherent weakness of many languages in failing to provide adequate data formatting functions should generate sufficient motivation for including sophisticated service drivers designed specifically for this purpose. For example, string manipulation routines, picture data statements and full screen text editing services may be some of the more eloquent features considered for inclusion as lower authority level primitives.

2. Related Research

Listed below is a sampling of topics that may be used for related research after construction of a working prototype:

1. Analysis of the impact on language development.
2. Analysis of the impact on integrated software packages development.
3. A study of the effects on concurrency issues.
4. Development of methods for data integrity protection.

D. A CLOSING REMARK

The interface based on a 'Dynamic Kernel' concept proposed in this thesis is not only conceptually feasible but, as it has been shown, is also implementable. Despite the many issues which may arise concerning its tendency to encourage subversion of current language design principles, the benefits realized by application programmers should stimulate sufficient interest towards its incorporation into present and future operating systems.

APPENDIX A
SUMMARY OF MS-DOS VER 2.0

A. OVERVIEW

1. DOS Structure

DOS Consists of the following four components:

a. Boot Record

The boot record resides on track 0, sector 1, side 0 of every disk formatted by the FORMAT command. It is put on all disks in order to produce an error message if the system is started with a non-DOS diskette in drive A. For fixed disks, it resides on the first sector (sector 1, head 0) of the first cylinder of the DOS partition.

b. BIOS

The Read-Only Memory (ROM) BIOS interface module (file IBMBIO.COM) provides a low-level interface to the ROM BIOS device routines.

c. DOS

The DOS program itself (file IBMDOS.COM) provides a high-level interface for user programs. It consists of file management routines, data blocking/ deblocking for the disk routines, and a variety of built-in functions accessible by user programs.

When these function routines are invoked by a user program, they accept high-level information via register and control block contents, then (for device operations) translate the requirement into one or more calls to IBMBIO to complete the request.

d. Command Processor

The command processor, COMMAND.COM, consists of four distinctly separate parts:

A resident portion resides in memory immediately following IBMDOS.COM and its data area. This portion contains routines to process interrupt types hex 22 (terminate address), hex 23 (CTRL- BREAK handler), and hex 24 (critical error handling), as well as a routine to reload the transient portion if needed. (When a program terminates, a checksum methodolcgy determines if the program had caused the transient portion to be overlaid. If so, it is reloaded.) All standard DOS error handling is done within this portion of COMMAND.COM. This includes displaying error messages and interpreting the reply of Abort, Retry, or Ignore.

An initialization portion follows the resident portion and is given control during startup. This section contains the AUTOEXEC file processor setup routine. The initialization porticn determines the segment address at which programs can be loaded. It is overlaid by the first program CCMMAND loads because it's no longer needed.

A transient portion is loaded at the high end of memory. This is (portion 3) the command processor itself, containing all of the internal command processors, the batch file processor, and (portion 4) a roution to load and execute external commands (files with filename extensions of .COM or .EXE). This loader is at the highest end of memory, and is invoked by the EXEC function call to load programs.

Portion 3 of COMMAND.COM produces the system prompt (such as A>), reads the command from the keyboard (or batch file) and causes it to be executed. For external commands, it builds a command line and issues an EXEC function call to load and transfer control to the program.

2. DOS Initialization

When the system is started (either System Reset or power ON with the DOS diskette in drive A), the boot record is read into memory and given control. It checks the directory to assure that the first two files listed are IBMBIO.COM and IBMDOS.COM, in that order. (An error message is issued if not.) These two files are then read into memory. (IBMBIO.COM must be the first file in the directory, and its sectors must be contiguous.)

The initialization code in IBMBIO.COM determines equipment status, resets the disk system, initializes the attached devices, causes device drivers to be loaded, and sets the low-numbered interrupt vectors. It then relocates IBMDOS.COM downward and calls the first byte of DOS.

As in IBMBIO.COM, offset 0 in DOS contains a jump to its initialization code, which is later overlaid by a data area and the command processor. DOS initializes its internal working tables, initializes interrupt vectors for interrupts hex 20 through hex 27 and builds a Program Segment Prefix for COMMAND.COM at the lowest available segment, then returns to IBMBIO.COM.

The last task of initialization is for IBMBIO.COM to load COMMMAND.COM at the location set up by DOS initialization. IBMBIO.COM then passes control to the first byte of COMMAND.

3. DOS Program Segment

When an external command or EXEC function call is made, DOS determines the lowest available address to use as the start of available memory for the program being invoked. This area is called the Program Segment (it must not be moved).

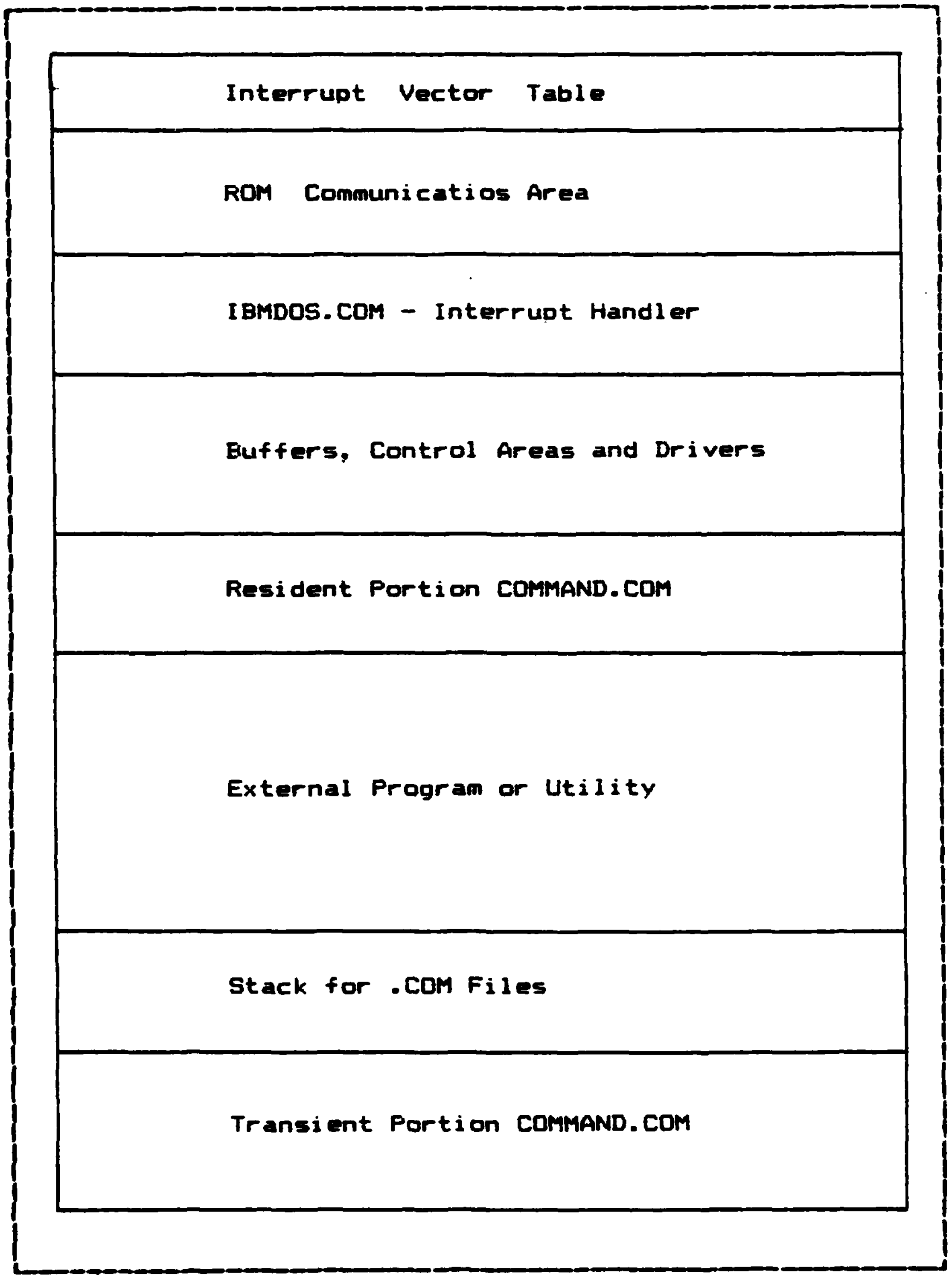

Interrupt Vector Table
ROM Communicatios Area
IBMDOS.COM - Interrupt Handler
Buffers, Control Areas and Drivers
Resident Portion COMMAND.COM
External Program or Utility
Stack for .COM Files
Transient Portion COMMAND.COM

Figure A.1 Memory Map of MS-DOS.

TABLE I

BIOS Interrupt Vectors

Interrupt No.		Name	Initialized by
0		Divide by zero	DOS
1		Single Step	DOS
2	8088	Non-Maskable	BIOS
3	Interrupt	Breakpoint	DOS
4	Vectors	Overflow	DOS
5		Print Screen	BIOS
6		Unused	---
7		Unused	---
8		8253 System Timer	BIOS
9		Keyboard	BIOS
A	8259	Unused	---
B	Interrupt	Unused	---
C	Vectors	Unused (Reserved)	---
D		Unused	---
E		Diskette	BIOS
F		Unused (Reserved)	---
10		Video I/O	BIOS
11		Equipment Check	BIOS
12		Memory Size	BIOS
13		Diskette I/O	BIOS
14	BIOS	Communications I/O	BIOS
15	Entry	Cassette I/O	BIOS
16	Points	Keyboard	BIOS
17		Printer I/O	BIOS
18	BICS	Cassette BASIC	BIOS
19	Entry	Power-on Reset	BIOS
1A	Points	Time of Day	BIOS
1B	User-Supplied	Keyboard Break	DOS
1C	Routines	Timer Tick	BIOS

TABLE II

BIOS Interrupt Vectors Cont.

Interrupt No.		Name	Initialized by
1D		Video Initialization	BIOS
1E	BIOS	Diskette Parameters	BIOS
1F	Parameters	Unused (Reversed)	---

TABLE III

Video I/O Operations (10 Interrupt)

(AH)	Operation	Additional Input Registers	Result Registers*
		CRT Interface routines	
0	Set video mode	(AL) = 0 40x25 B/W, Alpha (Default) = 1 40x25 Color, Alpha = 2 80x25 B/W,Alpha = 3 80x25 Color, Alpha = 4 320x200 Color, Graphics = 5 320x200 B/W, Graphics = 6 640x200 B/W, Graphics	None
1	Set cursor lines	CH Bits 0-4 = Start line for cursor CH Bits 5-7 = 0 CL Bits 0-4 = End line for cursor CL Bits 5-7 = 0	None
2	Set cursor position	(DH,DL)=Row,column (0,0) is upper left (BH)=Page number (0 for Graphics mode)	None
3	Read cursor position	(BH) = Page number (0 for Graphics mode)	(DH,DL) = Row,col. of cursor
4	Read light pen position	Ncne	(AH)=0 Light pen switch not down or not triggered (AH)=1 Valid light pen values in registers (DH,DL)=Row,column (CH)=Raster line (0-199) (BX)=Pixel column (0-319,639)

TABLE IV

Video I/O Operations (Type 10 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
5	Select active display page (Alpha modes)	(AL) = New page value (0-7 for Modes 0 and 1; 0-3 for Modes 2 and 3)	None
6	Scroll active page up	(AL) = Number of lines. Input lines blanked at bottom of window. (AL) = 0 blanks entire window (CH,CL) = Row, column of upper left corner of scroll (DH,DL) = Row, column of lower right corner of scroll (BH)=Attribute to be used on blank line	None
7	Scroll active page down	(AL) = Number of lines. Input lines blanked at top of window. (AL) = 0 blanks entire window. (CH,CL) = Row,column of upper left corner of scroll (DH,DL) = Row,column of lower right corner of scroll (BH) = Attribute to be used on blank line	None

TABLE V

Video I/O Operations (Type 10 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
	Character-Handling Routines		
8	Read attribute/character at current cursor position	(BH) = Display page (Alpha modes)	(AL) =Character read (AH) =Attribute of character read (Alpha modes)
9	Write attribute/character at current cursor position	(BH) = Display page (Alpha modes) (BL) = Attribute of character (Alpha) = Color of character (Graphics) (CX) = Count of characters to write (AL) = Character to write	None
10	Write character only at current cursor position	(BH) = Display page (Alpha modes) (CX) = Count of characters to write (AL) = Character to write	None
	Graphics Interface		
11	Set color palette (320x200 graphics)	(BH) = ID of palette color (0-127) (BL) = Color value to be used with that color ID	None

TABLE VI

Video I/O Operations (Type 10 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
12	Write dot	(DX) = Row number (CS) = Column number (AL) = Color value If Bit 7 of AL = 1, the color value is exclusive-ORed with the current contents of the dot	None
13	Read dot	(DX) = Row number (CS) = Column number	(AL) = Dot read
ASCII Teletype Routine for Output			
14	Write character to screen, then advance cursor	(AL) = Character to write (BL) = Foreground color (Graphics) (BH) = Display page (Alpha)	None
15	Read current video state	None	(AL) =Current mode See (AH)=0 for explanation (AH) =Number of character columns on screen (BH) =Current active display page

TABLE VII

Disk I/O Operations Type 13 Interrupt)

(AH)	Operation	Additional Input Registers	Result Registers*
0	Reset diskette system	None	None
1	Read diskette status	None	(AL) =Diskette status (see Figure 6-3)
2	Read sectors into memory	(DL) = Drive number (0-3) (DH) = Head number (0-1) (CH) = Track number (0-39) (CL) = Sector number (1-8) (AL) = Number of sectors (1-8) (ES:BX) = Address of buffer	(AL) =Number of sectors read CF Bit=0--Successful operation (AH)=0 CF Bit=1--Failed operation (AH)=Status (see Figure 6-3)
3	Write sectors from memory	Same as Read operation	Same as Read operation
4	Verify sectors	Same as Read operation, except (EX:BX) is not required	Same as Read operation
5	Format a track	(DL) = Drive number (0-3) (DH) = Head number (0-1) (CH) = Track number (ES:BX) = Sector information	Same as Read, except AL is not preserved

TABLE VIII

Disk I/O Operations (Type 13 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
0	Turn cassette motor on	None	None
1	Turn cassette motor off	Ncne	None
2	Read one or more 256-byte blocks from cassette	(CS) = Number of bytes to read (ES:BX) = Pointer to data buffer	(DX) =Number of bytes read (ES:BX)=Pointer to last byte read +1 CF Bit=0--Successful operation (AH) = 0 CF Bit=1--Error occurred (AH) =1--CRC error =2--Data transitions were lost =3--Data not found
3	Write one or more 256-byte blocks to cassette	(CS) = Number of bytes to write (ES:BX) = Pointer to data buffer	(CS) =0 (ES:BX)=Pointer to last byte written + 1

TABLE IX

Printer I/O Operations (Type 17 Interrupt)

(AH)	Cperation	Additional Input Registers	Result Registers*
0	Print a character	(AL) = Character to be printed (DX) = Printer tɔ be used (0-2)	(AH) =Status of operation (see Figure 6-4)
1	Initialize printer	(DX) = Printer to be used (0-2)	Same as print routine
2	Read printer status	(DX) = Printer to be used (0-2)	Same as print routine

TABLE X

DOS Interrupts

Interrupt Number	Name	Initialized to
20	Terminate Program	00B1:0022
21	Function Request	00B1:0015
22	Terminate Address	02F7:01FF
23	Ctrl-Break Exit Address	02F7:0204
24	Critical Error Handler	02B1:019B
25	Absolute Disk Read	0060:0015
26	Absolute Disk Write	0060:0018
27	Terminate, But Stay Resident	02B1:0187
28	Unused (Reserved)	---
29	Unused (Reserved)	---
2A	Unused (Reserved)	---
2B	Unused (Reserved)	---
⋮	⋮	⋮
3F	Unused (Reserved)	---

TABLE XI

Function Calls (Type 21 Interrupt)

(AH)	Operation	Additional Input Registers	Result Registers*
1	Wait for keyboard character, then display it (with Ctrl-Break check)	None	(AL) =Keyboard character
2	Display a character	(DL) = Display character	None
5	Print a character	(DL) = Print character	None
6	Read keyboard character, then, display it (without Ctrl-Break Check)	(DL) = OFFH	(AL) =Keyboard character, if available =0 if no character is available
6	Display a character	(DL) = Display character (value other than OFFH)	None

TABLE XII

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
7	Wait for keyboard character, but do not display it (without Ctrl-Break check)	None	(AL) =Keyboard character
8	Same as function 7, but with Ctrl-Break check	None	(AL) =Keyboard character
9	Display a string in memory	(DS:DX) = Address of string	None
A	Read key-board characters into a buffer	(DS:DX) = Address of buffer	None
B	Read key-board status (with Ctrl-Break check)	None	(AL) =OFFH if char-acter is available =0 if no char-acter is available

TABLE XIII

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
C	Clear keyboard buffer and call a keyboard input function	(AL) = Keyboard function number (1, 6, 7, 8, or A)	Per keyboard functicn
		Asynchronous Communications Functions	
3	Wait for asynchronous input character	None	(AL) =Asynchronous character
4	Output a character to asynchronous device	(DL) = Output character	None
		Disk Function	
D	Reset disk	None	None
E	Select default drive	(DL) = Drive number (0=A, 1=B)	(AL) =Number of drives in system (2 for single-drive system)

TABLE XIV

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
F	Open file	(DS:DX) = Address of unopened file control block (FCB)	(AL) =0 if file is found =OFFH if file not found
10	Close file	(DS:DX) = Address of opened FCB	Same as function F
11	Search for filename	(DS:DX) = Address of unopened FCB	(AL) =0 if filename found =OFFH if file-name not found
12	Find next occurrence of file-name	Same as function 11	Same as function 11
13	Delete file	Same as function 11	Same as function 11
14	Read sequen-tial file	(DS:DX) = Address of opened FCB	(AL) =0 if transfer successful =1 if no data in record =2 if insuf-ficient space
15	Write sequential file	Same as function 14	(AL) =0 if transfer is succussful =1 if disk is full =2 if insuf-ficient space

TABLE XV

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
16	Create a file	(DS:DX) = Address of unopened FCB	(AL)=0 if file is created =OFFH if no entry is empty
17	Rename a file	(DS:DX) = Address of filename to be renamed (DS:DX + 11H) = Address of new filename	(AL)=0 if rename is successful =OFFH if no match is found
19	Read default drive code	None	(AL)=Code of default drive (0=A, 1=B)
1A	Set disk transfer address	(DS:DX) = Disk transfer address	None
1B	Read allocation table address	None	(DS:DX)=Address of file allocation table (DX)=Number of allocation units (AL)=Records/allocation unit
21	Read random file	(DS:DX) = Address of opened FCB	Same as function 14
22	Write random file	Same as function 21	Same as function 15

TABLE XVI

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
23	Set file size	(DS:DX) = Address of unopened FCB	(AL)=0 if file size is set =OFFH if no matching entry is found
24	Set random record field	(DS:DX) = Address of opened FCB	None
26	Create a new program segment	(DX) = new segment number	None
27	Read random block	(DS:DX) = Address of opened FCB	(AL)=0 if transfer successful =1 if end-of-file =2 if wrap-around would occur =3 if last record is a partial record
28	Write random block	Same as function 27	(AL)=0 if transfer successful =1 if insufficient space

TABLE XVII

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
29	Parse a filename	(DS:SI) = Address of command line to parse (ES:DI) = Address of memory to be filled with an unopened FCB (AL) = 1 to scan off leading separators = 0 no scan-off	(AL) =0 if parse successful =1 if filename contains ? or * =OFFH if drive specifier is invalid
		Date and Time Functions	
2A	Get date	None	(CS) =Year (1980 - 2099) (DH) =Month (1 - 12) (DL) =Day (1 - 31)
2B	Set date	(CX) and (DX) = Date, in same format as function 2A	(AL) =0 if date is valid =OFFH if date is invalid
2C	Get time	None	(CH) =Hours (0 - 23) (CL) =Minutes (0-59) (DH) =Seconds (0-59) (DL) =1/100 Seconds (0 - 99)
2D	Set time	(CX) and (DX) = Time, in same format as function 2C	(AL) =0 if time is valid =OFFH if time is invalid

TABLE XVIII

Function Calls (Type 21 Interrupt) Cont.

(AH)	Operation	Additional Input Registers	Result Registers*
		Miscellaneous Functions	
0	Terminate program	None	None
25	Set interrupt vectors	(DS:DX) = Vector address (AL) = Interrupt type	None

TABLE XIX

Character Attributes (Ref: Interrupt 10)

The attribute of a character to be displayed is sent or recieved as follows:

```
  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

(Note: Black and White Mode only.)

Video Mode	Bit Position	Charactor Color	Background Color
Normal	B 0 0 0 i 1 1 1	White	Black
Reverse	B 1 1 1 i 0 0 0	Black	White
Non Display (Black)	B 0 0 0 i 0 0 0	Black	Black
Non Display (White)	B 1 1 1 i 1 1 1	White	White

Note: B	= 0	Non-blinking
	= 1	Blinking
i	= 0	Normal Intensity
	= 1	High Intensity

TABLE XX

Equipment Check (Type 11 Interrupt)

The equipment status is returned in the AX register as indicated below:

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |///|   |   |   |   |///|   |   |   |   |   |///|   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Bit/s	Meaning of Values Found
0	1 = diskette 0 = no diskette
2 - 3	0 0 = Unused 0 1 = 40 x 25 B/W, using color card 1 0 = 80 x 25 B/W, using color card 1 1 = 80 x 25 B/W, using B/W card
6 - 7	0 0 = 1 disk drive installed 0 1 = 2 disk drives installed 1 0 = 3 disk drives installed 1 1 = 4 disk drives installed (Note: relevant only if bit 0 = 1)
9 - 11	Number of RS-232 cards attached
12	1 = Game I/O 0 = no Game I/O
14 - 15	Number of printers attached

TABLE XXI

Diskette Status Byte (Ref: Interrupt Type 13)

The diskette status is retrieved in the AL register as indicated below:

```
   7     6     5     4     3     2     1     0
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Bit/s	Meaning of Values Found
0 - 1	01 = Invalid command or (if bit 3 = 1) attempt to transfer bata across 64K bcundry 10 = Address mark not found 11 = Attempt to write to write protected diskette
2	1 = Sector not found
3	1 = DMA overrun operation or (if bit 0 = 1) attempt to transfer data across 64K boundry
4	1 = Bad CRC on disk read
5	1 = Controller error
6	1 = Seek operation failed
7	1 = Drive failed to respond (time out error)

TABLE XXII

Keyboard I/O Status (Ref: Interrupt 16)

(AH)	Operation	Result Registers*

0	Scan code cf next key in buffer put in AH and character code into AL then the buffer paramenter is advanced	AH = scan code
		AL = character code

1	Returns status of buffer in the zero flag (ZF)	ZF = 1 buffer empty
		ZF = 0 buffer not empty

2	Returns keyboard status byte	AL = keyboard status

TABLE XXIII

Keyboard Status Bytes (Ref: Interrupt 16)

The keyboard status byte (KB_FLAG is retrived in AL register
(or can be found at location 18H if KB_FLAG_1 is desired)

```
   7     6     5     4     3     2     1     0
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Bit	Meaning of Values Found
KB_FLAG	
0	1 = right shift key depressed
1	1 = left shift key depressed
2	1 = control key depressed
3	1 = alternative key depressed
4	1 = scroll lock state on
5	1 = num lock state on
6	1 = caps state on
7	1 = insert mode on

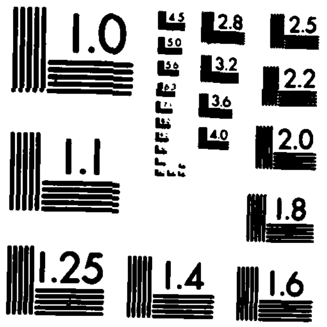
TABLE XXIV

Keyboard Status Bytes (Ref: Interrupt 16)

Bit	Meaning of Values Found
KB_FLAG_1	
0 - 2	Unused
3	1 = hold state on
4	1 = scroll lock key depressed
5	1 = num lock key depressed
6	1 = caps lock key depressed
7	1 = insert key depressed

AD-A146 584   A STANDARD OPERATING SYSTEM INTERFACE FOR MICROCOMPUTER   2/2
SOFTWARE DEVELOPMENT(U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA R STEMP MAR 84

UNCLASSIFIED   F/G 9/2   NL



ᐟOPY RESOLUTION TEST CHART

TABLE XXV

Printer I/O Status Byte (Ref: Interrupt 17)

The keyboard status is returned in the AH register as indicated below:

```
   7     6     5     4     3     2     1     0
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |/////|/////|     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Bit	Meaning of Values Found
0	1 = time out
3	1 = I/O error
4	1 = selected
5	1 = out of paper
6	1 = acknowledge
7	1 = busy

TABLE XXVI

Miscellaneous Interrupts

Type 12:	Determines number of 1K blocks of Read/Write memory that are installedin the System board The value is passed back in the AX register.
Type 19:	Performs a system warm boot
Type 1B:	Performs Ctrl-Break opeartion
Type 1C:	Can set vector at F0000:FF53 to perform any repeated task at 1/18.2 seconds

TABLE XXVII

Summary of DOS Commands

Note: In the column labeled Type, the I stands fcr Internal and the E stands for External

Command	Type	Purpose	Format
(Batch)	I	Executes batch file	<d:> file <parameters>
ECHO	I	Inhibits screen display	ECHO <ON/OFF message>
FOR	I	Interactive execution of commands	FOR %% variable IN <set> DO command
GOTO	I	Transfers control to line following label	GOTO label
IF	I	Conditional execution of commands	IF <NOT> condition command
SHIFT	I	Shift ccmmand lines	SHIFT
PAUSE	I	Provides a system wait	PAUSE <remark>
REM	I	Displays remarks	REM <remark>
ASSIGN	E	Routes requests to a different drive	ASSIGN <x = y <. . .>>
BACKUP	E	Backs up fixed disk files	BACKUP<d:><path><filename><.ext> d:</S></M></A></D:|mm-dd-yy>
BREAK	I	Checks for contrcl break	BREAK <ON/OFF>
CHDIR	I	Change current directory	CHDIR<<d:>path> or CD<<d:>path>
CHKDSK	I	Checks disk and reports status	CHKDSK <d:> <filename> </F> </V>

TABLE XXVIII

Summary of DOS Commands Cont.

Command	Type	Purpose	Format
CLS	I	Clears the display screen	CLS
COMP	E	Compares files	COMP <d:><path><filename<.ext>> <d:><path><filename<.ext>>
COPY	I	Copies files	COPY </A></B><d:><path>filename <.ext></A></B> <d:><path><filename<.ext>></A> </B></V> or COPY</A></B><d:><path>filename <.ext></A></B> <+<d:><path>filename<.ext></A> </B>...> <d:><path><filename<.ext>></A> </B></V>
DATE	I	Enter date	DATE <mm-dd-yy>
DIR	I	List filenames	DIR<d:><path><filename<.ext>> </P></W>
DISK-COMP	E	Compares diskettes	DISKCOMP <d:> <d:> </1> </8>
DISK-COPY	E	Copies diskettes	DISKCOPY <d:> <d:> </1>

TABLE XXIX

Summary of DOS Commands Cont.

Command	Type	Purpose	Format
ERASE	I	Deletes files	ERASE <d:><path><filename<.ext>> or DEL <d:><path><filename<.ext>>
FORMAT	E	Formats a diskette	FORMAT <d:></S></1></8></V></B>
GRAPH-ICS	E	Prints graphics display screen	GRAPHICS
MKDIR	I	Creates a sub-directory	MKDIR <d:> path or MD <d:> path
MODE	E	Set mode on printer/display	MODE LPT NO.:<n> <,<m><,P>> or MODE n or MODE <n> ,m <,T> or MODE COMn: baud <,parity <,databits <,P>>> or MODE LPT NO.:=COMn
PATH	I	Searches directories for commands or batch files	PATH <d:> <filename<.ext>> </T> </C></P>
PRINT	E	Queues and prints data files	PRINT <d:> <filename<.ext>> </T> </C> </P>. . .>
RECOVER	E	Recover files from disk or diskette	RECOVER<d:><path>filename<.ext> or RECOVER d:
RENAME	I	Renames files	REN<AME><d:><path>filename<.ext> filename<.ext>
RESTCRE	E	Restores diskette files to fixed disk	RESTORE d: <d:> <path> <file-name> <.ext> </S> </P>
RMDIR	I	Removes a sub-directory	RMDIR <d:>path or RD <d:>path

TABLE XXX

Summary of DOS Commands Cont.

Command	Type	Purpose	Format
SYS	E	Transfers DOS	SYS d:
TIME	I	Enter time	TIME <hh:mm:ss.xx>
TREE	E	Displays all directory paths	TREE <d:> </F>
TYPE	I	Displays file contents	TYPE <d:> <path> filename<.ext>
VER	I	Displays versicn number	VER
VERIFY	I	Verifies data	VERIFY <ON/OFF>
VOL	I	Displays volume identification	VOL <d:>

TABLE XXXI

Summary of Advanced DOS Commands

Command	Type	Purpose	Format
CTTY	I	Change to an auxiliary console	CTTY device-name
EXE2BIN	E	Converts .EXE files to .COM format	EXE2BIN <d:><path> <filename<.ext>> <d:><path><filename<.ext>>
FIND	E	Searches files for strings	FIND </V> </C> </N>string<d:> <path> filename<.ext>...>
MORE	E	Displays a screen full of data	MORE
PROMPT	E	Set new prompt	PROMPT <prompt-text>
SET	I	Inserts strings into command processor's environment	SET <name=<parameter>>
SORT	E	Sorts text data	SORT </R> </+n>

APPENDIX B

SUMMARY OF CP/M 80 VER 2.0

A. OVERVIEW

1. CP/M Structure

CP/M is logically divided into four distinct parts:

a. BIOS

The BIOS provides the primitive operations necessary to access the diskette drives and to interface standard peripherals (teletype, CRT, Paper Tape Reader/Punch, and user-defined peripherals), and can be tailored by the user for any particular hardware environment by 'patching' this protion of CP/M.

b. BDOS

The BDOS provides disk management by controlling one or more disk drives containing independent file directories. The BDOS implements disk allocation strategies which provide fully dynamic file construction while minimizing head movement across the disk during access.

c. CCP

The CCP provides symbolic interface between the user's console and the remainder of the CP/M system. The CCP reads the console device and processes commands which include listing the file directory, printing the contents of files, and controlling the operation of transient programs, such as assemblers, editors, and debuggers.

d. TPA

The last segment of CP/M is the area called the Transient Program Area (TPA). The TPA holds programs which are loaded from the disk under command of the CCP. During program editing, for example, the TPA holds the CP/M text editor machine code and data areas. Similarly, programs created under CP/M can be checked out by loading and executing these programs in the TPA.

2. <u>Functional Description</u>

The user interacts with CP/M primarily through the CCP, which reads and interprets commands entered through the console. The CCP addresses one of several disks which are online (the standard system addresses up to four different disk drives) and these are labelled A, B, C, and D. A disk is 'logged in' if the CCP is currently addressing the disk. In order to clearly indicate which disk is the currently logged disk, the CCP always prompts the operaor with the disk name followed the the symbol '>' indicating that the CCP is ready for another command. Upon initial start up, the CP/M system is brought in from disk.

All CP/M systems are initially set to operate in a 16K memory space, but can be reconfigured to fit any memory size on the host system. Following system signon, CP/M automatically logs in disk A, prompts the user with the symbol 'A>' (indicating that CP/M is currently addressing disk 'A'), and waits for a command. The commands are implemented at two levels: built-in commands and transient commands.

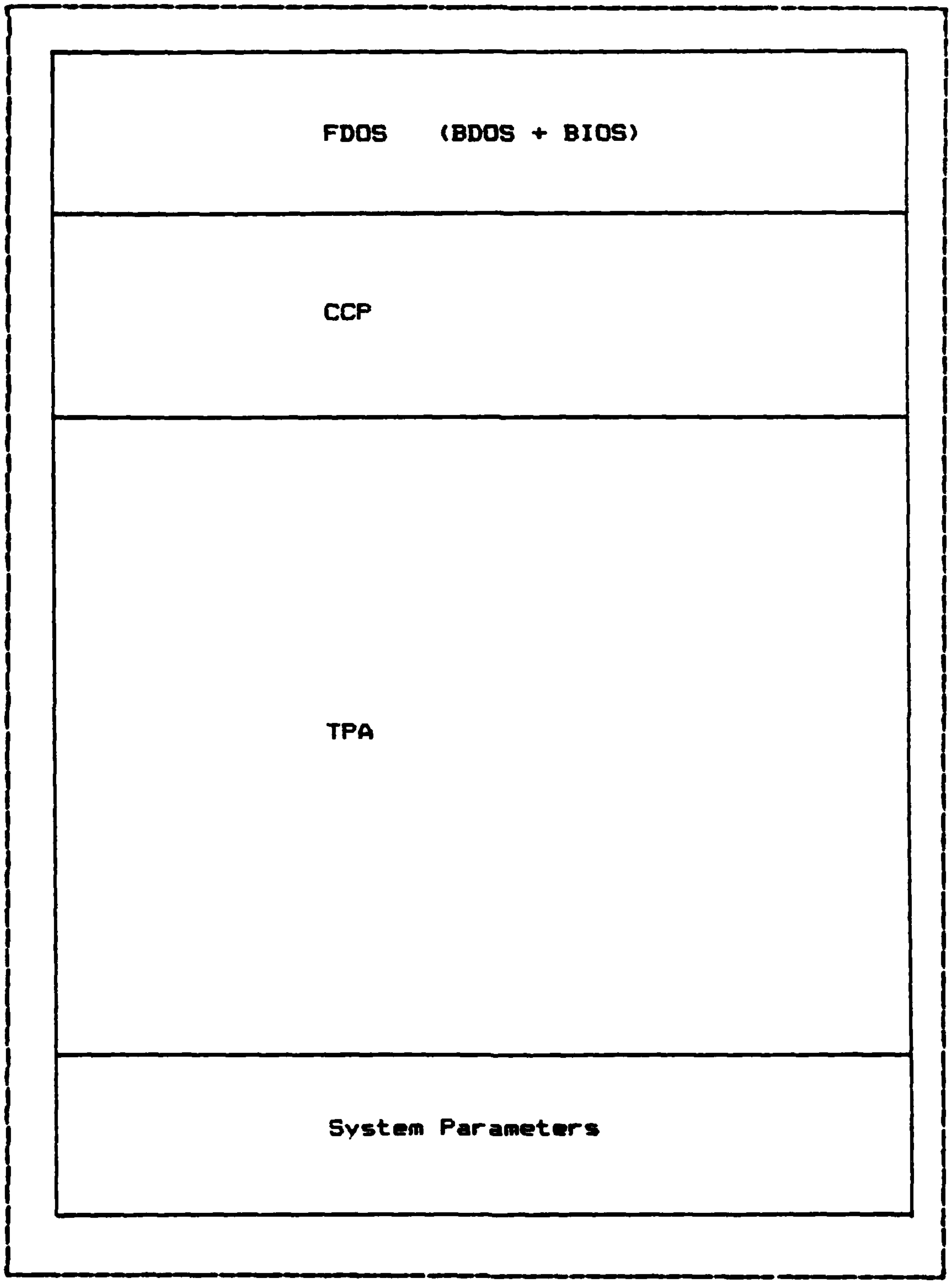


Figure B.1 Memory Map of CP/M 80.

TABLE XXXII

BIOS System Function Summary

(C)	Operation	Additional Input Registers	Result Registers*
0	System Reset	None	None
1	Console Input	Ncne	A = char
2	Console Output	E = char	None
3	Reader Input	None	A = char
4	Punch Output	E = char	None
5	List Output	E = char	None
6	Direct Console I/O	N/A	N/A
7	Get I/O Byte	Ncne	A = IOBYTE
8	Set I/O Byte	E = IOBYTE	None
9	Print String	DE = .Buffer	None
10	Read Console Buffer	DE = .Buffer	None
11	Get Console Status	None	A = 00/FF
12	Return Version Number	Ncne	HL = Version*

TABLE XXXIII

BIOS System Function Summary Cont.

(C)	Operation	Additional Input Registers	Result Registers*
13	Reset Disk System	Vcne	N/A
14	Select Disk	E = Disk Number	N/A
15	Open File	DE = .FCB	A = Dir Code
16	Close File	DE = .FCB	A = Dir Code
17	Search for First	DE = .FCB	A = Dir Code
18	Search for Next	None	A = Dir Code
19	Delete File	DE = .FCB	A = Dir Code
20	Read Sequential	DE = .FCB	A = Err Code
21	Write Sequential	DE = .FCB	A = Err Code
22	Make File	DE = .FCB	A = Dir Code
23	Rename File	DE = .FCB	A = Dir Code
24	Return Login Vector	None	HL = Login Vect*
25	Return Current Disk	Ncne	A = Cur Disk No.

TABLE XXXIV

BIOS System Function Summary Cont.

(C)	Operation	Additional Input Registers	Result Registers*
26	Set DMA Address	DE = .DMA	None
27	Get Addr (Alloc)	Ncne	HL = .Alloc
28	Write Protect Disk	None	N/A
29	Get R/O Vector	Ncne	HL = R/O Vect*
30	Set File Attributes	DE = .FCB	N/A
31	Get Addr (disk parms)	Ncne	HL = .DPB
32	Set/Get User Code	N/A	N/A
33	Read Random	DE = .FCB	A = Err Code
34	Write Random	CE = .FCB	A = Err Code
35	Compute File	DE = .FCB	r0, r1, r2
36	Set Random Record	LE = .FCB	r0, r1, r2

*Note that A=L and B=H upon return

TABLE XXXV

Summary of DOS Commands

Note: In the column labeled Type, the I stands for Internal and the E stands for External

Command	Type	Purpose	Format
DIR	I	Display directory of all files present on drive x	DIR x:<ce>
		Display directcry of files which match ambiguous file names and/or file extensions	DIR x:filename.typ<cr>
ERA	I	Erase the file 'filename.typ' on the disk in drive x	ERA x:filename.typ<cr>
		Erase all files on the diskette in the default drive	ERA x:*.*<cr>
REN	I	Finds the file oldname.typ and renames it newname.typ; the new name for the file is always to the left of the equal sign	REN newname.typ=oldname.typ<cr>
TYPE	I	Displays the contents of file filename.typ from drive x: on the console	TYPE x:filename.typ<cr>
SAVE	I	Save a portion of memory in a file, filename.typ, on drive x where nnn is a decimal number representing the number of pages of memory	SAVE nnn x:filetype.typ<cr>

TABLE XXXVI

Summary of DOS Commands Cont.

Command	Type	Purpose	Format
USER	I	Set the user number to n, where n is an integer decimal number from 0 to 15, inclusive	USER n<cr.>
x:<cr>	I	Log in another disk drive	x:<cr>
STAT	E	Displays the amount of free space available on the diskette in drive x:	STAT x:<cr>
		Displays the amount of space occupied by the file(s) filename.typ on drive x. The drive specifier x: is opticnal; if omitted, the current drive is assumed.	STAT x:filename.typ<cr>
PIP	E	Copies the file old.top on on drive y to the file new.typ on drive x using parameter(s) p	PIP x:new.typ=old.tcp(p)<cr>
		Creates a file new.fil on drive x which consists of files old1.fil and old2.fil, respectively, from drive y	PIP x:new.fil=y:old1.fil(p), old2.fil(p)<cr>
		Sends the contents of the file filename.typ on drive x to drive x to device dev:	PIP dev:=x:filename.typ(p)<cr>

TABLE XXXVII

Summary of DOS Commands Cont.

Command	Type	Purpose	Format
DUMP	E	Displays the hexadecimal representations of each byte stored in the file filename.typ on drive x	DUMP x:filename.typ<cr>
		Displays the Lexadecimal representations of the first file which matches the *.* parameters	DUMP x:*.*<cr>
SUBMIT	E	Creates a file $$$.SUB which contains the commands listed in filename.SUB and executes commands from this file rather than from the keyboard	SUBMIT filename<cr>
DISK-COPY	E	Transfer all information from one disk to another	A>DISKCOPY<cr>
SYSGEN	E	Places CP/M system on a disk	A>SYSBEN<CB>

APPENDIX C

GLOSSARY OF ABBREVIATIONS

ALI	-	Applicaticn Language Interface
BDOS	-	Basic Disk Operating System
BIOS	-	Basic Input/Output Services
BTP	-	Boot Time Processor
CCP	-	Console Command Proccessor
CRT	-	Cathode Ray Tube (Video Screen)
DBI	-	Data Block Interface
DBM	-	Data Block Manager
DEB	-	Data Exchange Block
DES	-	Data Encryption Standard
DOS	-	Disk Operating System
FCB	-	File Control Block
GKS	-	Graphics Kernal System
I/O	-	Input/Output
IPA	-	Index Paging Area
ISO	-	International Organization for Standardization
OS	-	Operating System
PC	-	Personal Computer
RAM	-	Random Access Memory
ROM	-	Read Only Memory

SD	-	Service Driver
SRI	-	Service Request Interface
SRM	-	Service Request Manager
SSD	_	System Services Directory
SSI	-	System Services Index
TPA	-	Transient Program Area

LIST OF REFERENCES

1. Cavanagh, Joseph J. F. Digital Computer Arithmetic Design and Implementation, McGraw Hill, 1983, pp. 446-448

2. Standard Specifications for Microprocessor Operating Systems Interfaces, IEEE Task 855 (Revision 5.0), undated

3. Graphics Kernel System (GKS) Functional Description, Draft International Standard ISO/DIS (January 1982)

4. National Bureau of Standards, Data Encryption Standard, January 1977, NTIS NBS-FIPS PUB 46

BIBLIOGRAPHY

Bourne, S.R., The UNIX System, Addison-Wesley Publishing Company, 1983

Calingaert, Peter, Operating System Elements, Prentice-Hall, 1982

Cowan, J., "Standard Operating System Interfaces", Sigsmall Newsl. (USA) Vol. 7, No. 3-4, December 1981

Ellis, J. R., A LISP Shell, Computer Science Department, Yale University, New Haven, Ct. 06520, undated

Hogan, Thom, Osborne CP/M User Guide, Osborne/McGraw-Hill, 1981

Irby, C., Bergsteinsson, L., Morgan, T., Newman, W., Tesler, T., A Methodology for User Interface Design, Xerox Palo Alto Research Center, 1977

Jacob, R. J. K., "Using Formal Specifications in the Design of a Human-Computer Interface", Communications of the ACM, vol. 26, No. 4, April 1983

Kaisler, Stephen H., The Design of Operating Systems for Small Computer Systems, John Wiley and Sons, 1983

MacLennan, Bruce J., Principles of Programming Languages: Design, Evaluation and Implementation, Holt, Rinehart and Winston, 1983

Martin, J., Design of Man-Computer Dialogues, Prentice-Hall, 1973

Mayer, R. E., "The Psychology of How Novices Learn Computer Programming", ACM Computing Surveys, vol. 13, No. 1, March 1981

Microsoft, Inc., Disk Operating System for the IBM Personal Computer, International Business Machines Inc., 1983

Miller, L. A. and Thomas, J. C. Jr., "Behavioral Issues in the Use of Interactive Systems", International Journal of Man-Machine Studies, vol. 9, No. 5, September 1977

Morgan, T. P., "An Applied Psychology of the User", ACM Computing Surveys, vol. 13, No. 1, March 1981

Mozelco, H., "A Human/Computer Interface to Accomodate Learning Stages", Communications of the ACM, vol. 24, No. 2, February 1982

Newman, W. M. and Sproul, R. F., Principles of Interactive Computer Graphics, McGraw-Hill, 1979

Norton, Peter, Inside the IBM PC: Access to Advanced Features and Programming, Robert J. Brady Co., 1983

Proceedings of the IFIP Working Conference on Command Languages, Command Languages, North-Holland Publishing Company, Amsterdam, 1975

Shneiderman, B., "Human Factors Experiments in Designing Interactive Systems", Computer, vol. 12, No. 12, December 1979

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	LT Roger Stemp, USN 209 Hart Drive Pensacola, Forida 32503	3
5.	Prof. Daniel Davis, Code 52Vv Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	Computer Technology Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943	1